

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER: H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees, H. Niemann,
D. Seitzer, B. Schmidt, H.J. Schneider, H. Wedekind

PARALLELISIEREN ORDNUNGSERHALTENDER PROGRAMMSYSTEME FÜR HIERARCHISCHE MULTIPROZESSORSYSTEME

Martin Rathke

DISSERTATION

BAND 18

NUMMER 2

ERLANGEN

MAI 1985

1985 #278



ISSN 0344 - 3515



Digitized by the Internet Archive
in 2026

PARALLELISIEREN ORDNUNGSERHALTENDER
PROGRAMMSYSTEME FÜR HIERARCHISCHE
MULTIPROZESSORSYSTEME

Der Technischen Fakultät
der Universität Erlangen-Nürnberg
zur Erlangung des Grades
DOKTOR - INGENIEUR

vorgelegt

von

Martin Rathke

ERLANGEN 1984

Als Dissertation genehmigt vom
Fachbereich Ingenieurwissenschaften
der Universität Erlangen-Nürnberg

Tag der Einreichung: 5. 11. 1984

Tag der Promotion: 28. 2. 1985

Dekan: Prof. Dipl.-Ing. G. Vetter

Berichterstatter: Prof. Dr. W. Händler

Dr. S. Keramidis

Mein besonderer Dank gilt

Herrn Prof. Dr. W. Händler für die Betreuung und Unterstützung dieser Arbeit.

Herrn Dr. S. Keramidis für die Übernahme des Zweitgutachtens

Herrn Dr. A. Bode, Herrn U. Hehn und Herrn Dr. J. Volkert für die Durchsicht der Arbeit und die vielen aus der Diskussion entstandenen konstruktiven Beiträge

Herrn A. Viegas de Vasconcelos für die fruchtbare Zusammenarbeit bei der Erstellung von Implementierungen zur Netzwerkanalyse

Frau L. Lange für die sorgfältige Erstellung der Zeichnungen und

allen Kolleginnen und Kollegen, die durch Kritik und Anregungen zum Gelingen dieser Arbeit beigetragen haben.

Inhalt

=====

1	Einleitung, Motivation	1
1.1	Ordnungserhaltende Programmsysteme	11
1.2	Hierarchische Multiprozessorsysteme mit eingeschränkten Ressourcen	15
1.3	Eine informelle Definition des Begriffs "parallele Programmierungsumgebung"	19
2	Vom sequentiellen Programm zu parallelen Verarbeitungsstrategien	21
2.1	Überlegungen und Methoden zur Parallelisierung von Programmen	21
2.1.1	Zur parallelen Ausführbarkeit von Aufgaben	22
2.1.2	Verfahren in idealen Rechensystemen	28
2.2	Zur Bildung von Teilaufgaben für Prozessoren eines Multiprozessorsystems, die E/A-Schnittstelle	34
2.3	Zur Teilung der Datenmenge	51
2.4	Die Entflechtung von Teilaufgaben	56
2.5	Zusammenfassung, Modultypen und Modulhierarchie	66
3	Parallele Verarbeitungsstrategien für ordnungserhaltende Programmsysteme	70
3.1	Die Zuordnung von Programmfunktionen zu Prozessoren eines hierarchischen Multiprozessorsystems	71
3.2	Die Aufteilung der Datenmenge auf Prozessoren	78
3.2.1	Der Raum möglicher Lösungen, optimale Lösungen	80
3.2.2	Eine Round-Robin-Heuristik	92
3.2.3	Eine zustandsorientierte Heuristik	96
3.2.4	Eine speicherorientierte Heuristik	98
3.3	Zur Frage einer geeigneten Rechnerarchitektur	103

4 Anwendungsbeispiele	108
4.1 Textformatierung	109
4.1.1 Programmstruktur und Alternativen zur Parallelisierung	110
4.1.2 Ergebnisse von Parallelisierungsstrategien	118
4.2 Netzwerkanalyse	125
4.2.1 Programmstruktur und Alternativen zur Parallelisierung	128
4.2.2 Ergebnisse von Parallelisierungsstrategien	134
5 Implementierungsaspekte	141
5.1 Struktur zugrundeliegender Programmierungsumgebungen	142
5.1.1 Elementare Hilfsmittel in einer EGPA-Umgebung	143
5.1.2 Elementare Hilfsmittel in einer UNIX ⁺ -Umgebung	145
5.1.3 Makro-Datenfluß	147
5.2 Kommunikation und Synchronisation	149
5.2.1 Möglichkeiten der Realisierung von Kommunikations- puffern	150
5.2.2 Zur Implementierung von Synchronisationsmechanismen	158
5.3 Erkennung und Beseitigung von Systemverklemmungen	170
5.3.1 Situationsbeschreibung, Grundlagen zu Systemverklemmungen in Betriebssystemen und Parallelprogrammen	171
5.3.2 Erkennung von Systemverklemmungen in Parallelprogrammen mit E/A-orientierter Kommunikation	177
5.3.3 Erkennung von Systemverklemmungen in Parallelprogrammen mit normierten Blockiert-Bedingungen	180
5.3.4 Erkennung von Systemverklemmungen in Parallelprogrammen mit beliebigen Blockiert-Bedingungen	184
5.3.5 Diagnose und Beseitigung von Systemverklemmungen, Stoppen von Prozessoren	192
5.4 Testmöglichkeiten für Parallelprogramme	195
5.4.1 Die Problematik von Testsystemen und Testausgaben in Parallelprogrammen	195
5.4.2 Zur Realisierung von Testsystemen für Parallelprogramme	197

⁺) UNIX ist ein Warenzeichen von Bell Laboratories.

6 Zum Entwurf einer E/A-orientierten parallelen Programmierungsumgebung	202
6.1 Struktur von Parallelprogrammen, Gestaltungsgrundlagen	202
6.2 Sprachmittel, Einbettung in Organisationsformen für Multiprozessorsysteme	205
7 Zusammenfassung und Ausblick	219
Literaturverzeichnis	225
Stichwortverzeichnis	241

1 Einleitung, Motivation

Die Zeit, die ein einzelner Rechner benötigt, um eine ihm zugeteilte Aufgabe ausführen zu können, läßt sich in der Regel verkürzen, wenn die Bearbeitung der Aufgabe unter mehreren Rechnern aufgeteilt wird, die zu einem System zusammengeschaltet sind.

Diese Überlegung ist die wohl am meisten einleuchtende unter denen, die den Entwurf und die Entwicklung von Multiprozessor-Rechenanlagen vorangetrieben haben und letztlich auch rechtfertigen.

Dabei ist jedoch noch keine befriedigende Lösung zu dem Problem gefunden, wie Programme auf Multiprozessorsystemen zu erstellen sind, die

1. die erwünschte Leistungsverbesserung bringen und dabei
2. nicht komplizierter als Programme für den Monorechner sind.

Gerade dieser Aspekt dürfte ausschlaggebend dafür sein, daß Multiprozessorsysteme noch kaum kommerziell vertrieben werden.

Die vorliegende Arbeit beschäftigt sich mit Fragen der Erstellung von Programmen, die auf Multiprozessoranlagen ablaufen sollen, d. h. Fragen, die unter dem Begriff "Parallelisieren von Programmen" zusammengefaßt werden können.

Die Vorgehensweise wird dabei (im Gegensatz zu den meisten Arbeiten, die sich mit diesem Themenkomplex beschäftigen) weniger davon bestimmt, mit der Parallelisierung eines Programms für ein Multiprozessorsystem zu einer Lösung mit minimaler Gesamtausführungszeit¹⁾ zu kommen. Es stehen vielmehr Fragen der Programmierbarkeit im Vordergrund. Ziel der durchgeführten Un-

1) Die Frage, wie eine Aufgabe auf einem Multiprozessorsystem zu implementieren ist, um eine Lösung mit minimaler Gesamtausführungszeit zu erhalten, führt auf das Multiprozessor-Scheduling-Problem, welches zur Klasse der NP-vollständigen Probleme gehört (vgl. hierzu auch Kap. 3.2.1).

tersuchungen ist es, eine Systematik zur Entwicklung eines Parallelprogramms aufzuzeigen, die sicherstellt, daß

1. die durch das Parallelprogramm erzielten Ergebnisse im Sinn der zu lösenden Aufgabenstellung korrekt sind,
2. das Parallelprogramm für beliebige Abläufe fehler- und verklemmungsfrei ist und
3. eine dem Aufwand an verfügbarer Hardware angemessene Leistungssteigerung²⁾ gegenüber einem sequentiellen Programm zur Lösung der gleichen Aufgabe bietet.

Während die ersten beiden Forderungen als grundsätzlich für verschiedene Fassungen eines die gleiche Aufgabe realisierenden Programms angesehen werden können, führt Forderung drei auf eine typische im Zusammenhang mit der Programmierung von Anwendungen auf Multiprozessorsystemen bestehende Fragestellung.

Es wäre gegenwärtig vermessen, die angeschnittenen Problemstellungen mit einem Programm lösen zu wollen, welches als Eingabe die zu lösende Aufgabe und das verfügbare Multiprozessorsystem erhält und als Ausgabe ein Parallelprogramm liefert, welches den geforderten Kriterien genügt.

Es läßt sich zwar eine Entwicklung von Programmiersprachen denken, die stärker als die bestehenden Sprachen eine Denkweise in parallelen Abläufen suggerieren (vgl. etwa die APL-Notation für Vektoren und die darauf definierten "parallelen" Verknüpfungsoperationen) oder von Compilern, die ein sequentielles Programm in parallel ausführbare Teilaufgaben übersetzen. Mit solchen Werkzeugen könnte der Vorgang der Parallelisierung von Programmen für spezielle Multiprozessorsysteme weitgehend automatisiert werden.

- 2) Eine dem Aufwand an bereitgestellter Hardware entsprechende Leistungsverbesserung liegt vor, wenn das Verhältnis zwischen Laufzeit des Programms auf dem Monoprozessor und Laufzeit des Programms auf dem Multiprozessor-System nicht wesentlich kleiner als die Zahl der vorhandenen Prozessoren ist.

Doch solange Programme mit bestehenden - aus einer "sequentiellen" Denkweise geprägten - Programmiersprachen erstellt werden, muß ein Parallelisieren von Programmen mit Methoden erfolgen, die durch die folgenden Schlagworte charakterisiert werden können:

1. Partitionierung des Programms in Teilaufgaben
2. Zuordnen von Teilaufgaben zu Prozessoren (unter Beachtung der Verbindungsstruktur der Prozessoren und der durch Kontrollfluß und Datenfluß gegebenen Abhängigkeit der Teilaufgaben)
3. Erstellung der Implementierung unter Beachtung der vom Betriebssystem vorgesehenen Mechanismen zur Prozessorkommunikation

Die vorliegende Arbeit stellt ein Verfahren vor, diese Aspekte der Parallelisierung, - ohne die Entwicklung "neuer" Sprachkonstrukte - systematisch aus einem sequentiellen Programm zu entwickeln. Im Endergebnis ergibt sich eine parallele Version, für die das sequentielle Programm als Spezialfall angesehen werden kann.

Dabei wird angestrebt, so viele Teile wie möglich aus dem sequentiellen Programm unverändert in die parallele Lösung zu übernehmen. Insbesondere wird ein Weg aufgezeigt, die Zuordnung von (parallel ausführbaren) Teilaufgaben zu Prozessoren und die Implementierung der Prozessorkommunikation aus dem Benutzeralgorithmus heraus in eine parallele Programmierungsumgebung - eine Art "Systemebene" - zu verlagern.

Die Vorteile, die ein solcher Ansatz bietet, sind:

1. Sequentielle und parallele Programme werden fast gleich (vgl. Mythen der Parallelprogrammierung in [JoSch80]).
Die Entwicklung von Parallelprogrammen kann von Benutzerseite mit den Programmierhilfsmitteln (Compiler/Codegenerator und Binder) durchgeführt werden, die ihm vom Mono-Betriebssystem zur Verfügung gestellt werden.

2. Die Suche nach Programmfehlern kann größtenteils im sequentiellen Programm auf dem Monorechner erfolgen, wo im allgemeinen eine "besser" zu handhabende Testumgebung als auf dem Multiprozessorsystem zur Verfügung steht.

In Fehlerfällen, die auf spezifische "Parallelteile" zurückzuführen sind, können Diagnosen weitgehend von der Programmierungsumgebung durchgeführt werden.

Damit ergeben sich vor allem in der Programmentwicklungszeit Vorteile, sodaß ein solches Vorgehen auch als Ausgangspunkt "effizienter" gestalteter Lösungen verwendet werden kann.

3. Parallelprogramme, die nach solchen Grundsätzen erstellt werden, lassen sich leichter zwischen verschiedenen Multiprozessorsystemen portieren. Unter Umständen reicht es aus, die Module, in denen die Prozessorkommunikation niedergelegt ist, auszutauschen.

Die Philosophie des Ansatzes läuft allerdings einer Tendenz entgegen, die bei der Entwicklung vieler paralleler Algorithmen³⁾ in der Vergangenheit verfolgt wurde: der Konstruktion eines völlig neuen parallelen Algorithmus, obwohl zur Lösung der gestellten Aufgabe ein sequentieller Algorithmus existierte. Dies geschah hauptsächlich deshalb, weil eine Parallelisierung auf der Basis einer vorhandenen sequentiellen Lösung im Multiprozessorsystem keinen akzeptablen Geschwindigkeitsgewinn⁴⁾ erwarten ließ.

Es ist nicht die Absicht dieser Arbeit, die Berechtigung "direkter", "systemspezifischer" Implementierungen oder gegebenenfalls die Notwendigkeit der Konstruktion "neuer" paralleler Algorithmen in Frage zu stellen. Es läßt sich im allgemeinen auch

3) Hinweise auf Übersichtsarbeiten können etwa [HosW83] entnommen werden.

4) Hier wie auch im folgenden soll unter dem Begriff "Geschwindigkeitsgewinn" der Quotient aus der Zeit zur Ausführung der Aufgabe auf einem Monoprozessor und der Zeit zur Ausführung der Aufgabe auf einem Multiprozessorsystem verstanden werden.

erwarten, daß system- und anwendungsspezifische Verfahren der Parallelisierung einen höheren Geschwindigkeitsgewinn gegenüber einem sequentiellen Programm erwarten lassen als eine Implementierung nach dem hier vorgeschlagenen Ansatz.

Es muß dabei allerdings bedacht werden, daß Aspekte der "einfachen" Kontrollierbarkeit von Programmen mit Zunahme der Programmkomplexität eine entscheidende Rolle bei der Programmerstellung spielen. Zumal für Programme mit mehreren tausend Programmzeilen besteht der überwiegende Teil der Programmerstellung aus der Suche nach Programmfehlern. Und: Die Konstruktion völlig neuer paralleler Algorithmen für eine spezielle Problemstellung verlangt im Gegensatz zum empfohlenen Ansatz vom Benutzer eine Art "Kreativität", die sich kaum zu einer generell anwendbaren Systematik bei der Parallelisierung erweitern läßt.

Dem Aspekt der Programmierbarkeit paralleler Algorithmen ist erst Bedeutung zugekommen, seit Anwendungen auf realen Multiprozessor-systemen implementiert worden sind. Seit 1978 ist in Erlangen ein System aus fünf AEG 80/60-Rechnern installiert, die in Form einer Pyramide - der EGPA⁵⁾-Pilotpyramide - miteinander gekoppelt sind. Die Erfahrungen mit vielen Implementierungen auf dem EGPA-System haben gezeigt, daß die Befolgung der in den nachfolgenden Kapiteln beschriebenen Methode der Parallelisierung und die damit gegebenen Möglichkeiten, "parallelspezifische" Probleme in eine "Systemebene" zu verlagern, entscheidend die Entwicklung korrekter und bezgl. der Laufzeit zufriedenstellender Parallel-Implementierungen beschleunigt hat.

Wenn Programme für das Multiprozessorsystem - wie hier empfohlen - aus sequentiellen Programmen entwickelt werden, ergeben sich etwa Fragen der folgenden Art:

1. Nach welchen Kriterien können sequentielle Programme parallelisiert werden?

An welchen Stellen im Programm ist dabei mit einer Parallelisierung anzusetzen?

5) Erlangen General Purpose Array.

2. Welche Kriterien müssen bei der Erstellung sequentieller Programme beachtet werden, um einen möglichst einfachen Übergang zu parallelen Programmen zu erreichen?
3. Zu welchen Algorithmen kommt man bei der Parallelisierung? Nach welchen Kriterien können hierbei Aufgaben und Daten auf verschiedene Prozessoren verteilt werden? Gibt es hierzu eine zufriedenstellende Strategie? Wie kann gegebenenfalls die Optimalität einer gefundenen Lösung nachgewiesen werden?

Antworten auf diese Fragen werden in der Literatur in der Regel nur unvollkommen gegeben. Zudem gibt es sehr wenige aus der Praxis entstandene Erfahrungsberichte, die Entwurf und Implementierung von Algorithmen auf Multiprozessoranlagen beschreiben und daraus Methoden zu zufriedenstellenden Implementierungen ableiten.

Es ist auch sicher nicht möglich, obige Fragen generell für beliebige Klassen von Programmsystemen und beliebige Konfigurationen von Multiprozessorsystemen in Form eines "Kochrezepts" zum Vorgehen bei der Parallelisierung zu beantworten.

Um dennoch zu einer Systematik bei der Parallelisierung zu kommen, die sich auf ein möglichst breites Spektrum von Aufgabenstellungen anwenden läßt, wurde die Untersuchung auf einen als "schwierig parallelisierbar" geltenden Typ von Aufgaben konzentriert, den der ordnungserhaltenden Programmsysteme.

Ordnungserhaltende Programmsysteme sind alle Arten von Compilern und Interpretern. Sie unterscheiden sich von anderen Programmsystemen dadurch, daß nur dann im Sinne der Aufgabenstellung korrekte Ergebnisse erzielt werden können, wenn zu verarbeitende Daten in einer fest vorgeschriebenen (sequentiellen) Reihenfolge eingelesen und erzielte Ergebnisse in einer fest vorgeschriebenen (sequentiellen) Reihenfolge wieder ausgegeben werden. Sie "eignen" sich deshalb für eine Parallelisierung nach dem empfohlenen Ansatz, da bei diesen Programmsystemen in der Regel aufgrund von Umfang und Komplexität der sequentiellen Programmpakete vor Modifikationen am Algorithmus zum Zweck der Parallelisierung

zurückgeschreckt wird, d. h. die Erstellung völlig neuer paralleler Algorithmen nicht ernsthaft erwogen wird.

Hierarchische Multiprozessorsysteme wurden als Zielrechnersystem in erster Linie deshalb ausgesucht, da die verfügbare hierarchische EGPA-Struktur Grundlage der Implementierungen war. Es ist darüberhinaus aber eine verbreitete Ansicht, daß die Organisation von Programmen und Systemen als Hierarchie der "beste" Weg ist, sowohl Multiprozessorsysteme aufzubauen als auch Parallelprogramme in einer "überschaubaren" Art zu implementieren (vgl. etwa [Ens77], [FKLV83], [HHS76], [Hän82], [HHHS83], [Wal82], [Wen78]). Die Vorgehensweise bei der Parallelisierung und die daraus resultierende Programmstruktur stellen jedoch den Anspruch, daß sie sich ohne Schwierigkeiten auf andere Konfigurationen von Multiprozessorsystemen übertragen lassen.

Grundlage der nachfolgenden Ausführungen sind von Fromm [From82] und Kneißl [Knei82] an EGPA vorgenommene Untersuchungen zur Erstellung von Programmen für Multiprozessor-Rechenanlagen.

Fromm [From82] beschreibt in seiner Dissertation Hilfsmittel und Methoden, welche dazu dienen, zu einem in geeignete Teilaufgaben zerlegten Algorithmus und einer gegebenen Maschinenstruktur eine optimale Zuordnung der Teilaufgaben zu den verfügbaren Prozessoren zu bestimmen. Die Funktionsfähigkeit der eingeführten Methoden wird dort u. a. an einem Beispiel für die EGPA-Pilotpyramide nachgewiesen.

Mit der hier eingeführten Systematik der Parallelisierung wird untersucht, wie von der Seite einer parallelen Programmierungsumgebung - ohne Änderungen am Benutzeralgorithmus - eine Annäherung an optimale Aufgabenverteilungen für die Prozessoren eines Multiprozessorsystems unterstützt werden kann.

Kneißl [Knei82] dokumentiert in seiner Dissertation ein auf dem Datenflußkonzept aufbauendes System, welches Mechanismen paralleler Verarbeitung mit wenigen Begriffen wie Funktion, Argument, Ergebnis, Verfügbarkeit eines Ergebnisses zu beschreiben gestattet.

Mit der empfohlenen Systematik zur Parallelisierung und den dabei entstehenden parallelen Programmstrukturen wird ein Weg aufgezeigt, Parallelprogramme ohne Änderungen am Benutzeralgorithmus in verschiedene Programmierumgebungen für Multiprozessorsysteme mit unterschiedlichen Kommunikationstechniken umzusetzen, z. B. in Programme, die die in [Knei82] beschriebenen Makro-Datenflußmechanismen verwenden.

Der Gesichtspunkt einer quasi als Berechnungsvorschrift formulierbaren Vorgehensweise bei der Umsetzung von Parallelprogrammen in verschiedene Systemumgebungen⁶⁾ wird dann eine wesentliche Rolle spielen, wenn Parallelprogramme von einem Multiprozessorsystem auf ein anderes Multiprozessorsystem zu portieren sind. In Zukunft - man denke etwa an die Erstellung von "parallelen" Programmbibliotheken - wird man es sich kaum mehr erlauben können, Algorithmen für jedes "neue" System "neu" zu programmieren.

Die Gliederung der vorliegenden Arbeit ist von der informell eingeführten Methodik der Parallelisierung unter dem Aspekt der Programmierbarkeit paralleler Lösungen und den hierbei zu untersuchenden Aufgabenbereichen geprägt.

Nach einer Präzisierung der Begriffe "ordnungserhaltende Programmsysteme", "hierarchische Multiprozessorsysteme" und "Programmierungsumgebung" wird im zweiten Kapitel von aus der Literatur bekannten Ansätzen zur Parallelisierung von Programmen ausgehend eine Methode entwickelt, ordnungserhaltende sequentielle Programmsysteme in eine für die Implementierung auf dem betrachteten Zielrechnersystem geeignete Struktur umzuformen.

6) Dabei muß im allgemeinen beachtet werden, daß in Parallelprogrammen der Wechsel auf eine neue Kommunikationstechnik zu unterschiedlichem (zeitlichen) Prozeßverhalten führt, welches den erreichbaren Geschwindigkeitsgewinn beeinflusst (vgl. Kap. 5.2).

Im dritten Kapitel werden von dieser Programmstruktur ausgehend parallele Verarbeitungsstrategien für ordnungserhaltende Programmsysteme zusammengestellt. Dabei wird nach den beiden wichtigsten Ansätzen zur Parallelisierung, nämlich "Aufteilung von Programmfunktionen auf Prozessoren" und "Aufteilung der zu bearbeitenden Datenmenge auf Prozessoren" unterschieden. Dies schließt auch die Frage der Annäherung an optimale Lösungen bzw. an optimal geeignete Rechnerstrukturen ein.

Im vierten Kapitel werden Methode und parallele Verarbeitungsstrategien der Kapitel zwei und drei auf die beiden Anwendungsbeispiele Textformatierung und Netzwerkanalyse angewendet. Die Ergebnisse der auf EGPA durchgeführten Implementierungen erlauben nicht nur eine quantitative Bewertung verschiedener Problemlösungen, sondern auch einen quantitativen Vergleich des eingeschlagenen Weges mit "direkten" Methoden.

Im fünften Kapitel wird auf Implementierungsaspekte der erstellten Parallel-Programme eingegangen. Grundlage hierfür sind in EGPA und UNIX realisierte Programmierungsumgebungen, die stellvertretend für verschiedene Ansätze der Realisierung eines Zusammenwirkens von Prozessoren eines Multiprozessorsystems stehen.

Im Zusammenhang des Austauschs von Informationen zwischen verschiedenen Prozessoren sind Fragen der Verwaltung des nur in eingeschränkter Menge für die Prozessoren zugreifbaren Speichers zu lösen. Hierzu werden verschiedene Möglichkeiten untersucht.

Speziell in Programmen, in denen mehrere Prozessoren auf gemeinsam benutzte Datenbereiche zugreifen, ergeben sich Probleme der Koordination und damit der Synchronisation. Es wird gezeigt, wie Aspekte der Synchronisation und Kommunikation so in die entstandenen Parallelprogramme integriert werden können, daß die zugrundeliegenden sequentiellen Programme (für die derartige Überlegungen nicht erforderlich sind) in dieser Beziehung als Spezialfälle beliebiger paralleler Programmfassungen erscheinen. Ferner wird deutlich, auf welcher Ebene im Programm generell die

Untersuchung verschiedener Koordinierungsstrategien vorgenommen werden sollte. Für einige der aus der Literatur bekannten Strategien zur Realisierung von Prozessorkommunikation werden Implementierungsvorschläge zur Anpassung an die definierte Schnittstelle gegeben.

Wenn von Parallelprogrammen im "Entwicklungsstadium" ausgegangen wird, können Programmierfehler nicht ausgeschlossen werden. Wenn aufgrund eines solchen Fehlers Prozessoren ausfallen, können Systemverklemmungen (Deadlocks) die Folge sein. Aus dieser Notwendigkeit heraus erschien es wichtig, zu untersuchen, wie aus der Theorie der Betriebssysteme bekannte Algorithmen zur Erkennung, Diagnose und Beseitigung von Deadlocks in die Konstruktion von Parallelprogrammen einfließen können. Die Analyse und Beseitigung von Fehlern in Programmen, an deren Ausführung mehrere Prozessoren beteiligt sind, ist problematisch, wenn ein hierdurch bedingter Ausfall eines Prozessors anderen Prozessoren "mitgeteilt" werden muß und diese ihrerseits "angemessen" darauf reagieren sollen. Es zeigt sich jedoch, daß für die entwickelten Parallelprogramme über eine geeignet definierte parallele Programmierungsumgebung Hilfsmittel bereitgestellt werden können, mit denen die Behandlung von Fehlern in Parallelprogrammen weitgehend auf die Problematik der Fehlerbehandlung in sequentiellen Programmen zurückführbar ist.

Im sechsten Kapitel wird schließlich als Konsequenz aus der vorgeschlagenen Systematik zur Parallelisierung und den daraus implementierten Parallelprogrammen eine "passende" parallele Programmierungsumgebung entworfen und Möglichkeiten ihrer Einbettung in verschiedene Organisationsformen für Multiprozessor-Betriebssysteme diskutiert.

Die vorliegende Arbeit ist direkt aus praktischen Erfahrungen mit der Erstellung paralleler Algorithmen für unterschiedliche Anwendungen entstanden. Sie zeigt eine Möglichkeit, die Lücke zwischen theoretischen Untersuchungen über Parallelverarbeitung und praktischer Anwendung der erzielten Ergebnisse zur Erstellung fehlerfreier und "schnellerer" paralleler Programme zu verkleinern.

1.1 Ordnungserhaltende Programmsysteme

Wenn die Leistung existierender Multiprozessor-Systeme dokumentiert wird, geschieht dies unter anderem dadurch, daß für ausgesuchte Programme (Benchmarks) ein Laufzeitvergleich zwischen einer Version für den Monoprozessor und einer auf das betrachtete Multiprozessor-System zugeschnittenen Version durchgeführt wird.

In der Regel werden zu diesem Zweck Anwendungen aus der numerischen Mathematik verwendet. Dafür gibt es eine Reihe von Gründen, von denen hier nur drei der wichtigsten genannt werden sollen:

1. Viele auf Rechenanlagen implementierten Anwendungen - vor allem aus dem Bereich der Naturwissenschaften - lassen sich mit den Methoden der numerischen Mathematik lösen bzw. auf Probleme in diesem Bereich zurückführen.
2. Bei derartigen Problemstellungen handelt es sich im allgemeinen um Aufgaben, die einen (im Verhältnis zur benötigten Zeit für Ein- und Ausgaben) sehr hohen Anteil an CPU-Zeit verbrauchen und damit bei einer Vervielfachung der durch die Hardware zur Verfügung gestellten Rechenleistung eine dem Aufwand entsprechende Leistungsverbesserung erwarten lassen.
3. Algorithmen der numerischen Mathematik haben häufig eine Struktur derart, daß Zwischenergebnisse und Ergebnisse in Form von Vektoren oder Matrizen entstehen, deren einzelne Elemente nach Vorliegen der benötigten Eingaben unabhängig voneinander berechnet werden können (vgl. hierzu Kap. 4.2). Dies legt schon ausgehend von den verwendeten Datenstrukturen eine Verteilung der Aufgaben über eine Aufteilung der Datenstruktur auf die verschiedenen Prozessoren nahe und ist damit einer Parallelisierung "leicht" zugänglich.

Wenn sich jedoch Multiprozessor-Systeme am Markt durchsetzen sollen, kann man sich nicht allein auf die Implementierung "leicht" zu parallelisierender Programme zurückziehen. Eine Chance wird nur bestehen, wenn es gelingt, beliebige existierende "große" Programmsysteme dem Multiprozessor-System anzupassen,

d. h. zu parallelisieren und damit einen signifikanten Geschwindigkeitsgewinn in der Laufzeit gegenüber den entsprechenden Programmsystemen für den Monoprozessor zu erzielen.

Diese Überlegung ist Ausgangspunkt zur Definition einer Klasse von Programmen, die landläufig als "scheinbar nicht parallelisierbar" gelten.

Schauen wir uns dazu einige Beispiele an:

1. Schreiben eines Textes auf einen Fernschreiber
2. Kopieren einer (Eingabe-)Datei in eine (Ausgabe-)Datei über einen E/A-Kanal
3. Berechnung des Funktionswertes eines gegebenen Polynoms an einer bestimmten Stelle mit Hilfe des Hornerschemas (vgl. [Kuck77])
4. Berechnung von x^{2^k} für ein gegebenes x und ein gegebenes k (vgl. [HosW83])

Im ersten Fall ist eine Parallelisierung unmöglich, wenn die (eigentlich selbstverständliche) Voraussetzung gemacht wird, daß zur gleichen Zeit nicht mehr als ein Zeichen ausgegeben werden kann. Im zweiten Fall ist eine Parallelisierung unmöglich, wenn nicht mehr als ein Satz gleichzeitig gelesen bzw. geschrieben werden kann. In beiden Fällen wird eine theoretisch denkbare Parallelarbeit durch Zahl bzw. Betriebsart der verwendeten Hardwarebausteine verhindert. In den Fällen drei und vier liegt die Nicht-Parallelisierbarkeit jedoch in der "sequentiellen" Natur des Problems bzw. seiner Lösung begründet.

Es sollte an dieser Stelle darauf hingewiesen werden, daß der hier intuitiv verwendete Begriff "nicht parallelisierbar" nichts mit der Zahl der Prozessoren zu tun hat, aus denen das zugrundeliegende System besteht. Es ist lediglich so, daß beschriebene Probleme mit einem Multiprozessorsystem nicht schneller als mit einem Monoprozessorsystem lösbar sind.

Welche Möglichkeiten bestehen nun, für diese Beispiele durch Implementierung der Aufgabe auf einem Multiprozessorsystem einen Geschwindigkeitsgewinn zu erzielen?

Im ersten Fall ist ein Geschwindigkeitsgewinn durch Parallelarbeit möglich, wenn die Arbeit der Ausgabe von mehreren peripheren Einheiten durchgeführt wird, wobei jede nur für einen Teil des betroffenen Textes zuständig ist. In diesem Fall wird die Menge der zu bearbeitenden Daten auf Arbeitseinheiten aufgeteilt.

Im zweiten Fall ergibt sich ein Geschwindigkeitsgewinn durch Parallelarbeit, wenn Eingabe und Ausgabe von verschiedenen Einheiten durchgeführt werden. In diesem Fall kann man von einer Aufteilung von Funktionen sprechen.

Im dritten Fall [Kuck77] ergibt sich ein Geschwindigkeitsgewinn, wenn zur Polynomberechnung statt des Hornerschemas eine andere Berechnungsmethode verwendet wird, also ein neuer Algorithmus erstellt wird. In diesem Fall könnte man von einer Parallelisierung auf der Ebene des Algorithmus sprechen.

Dagegen läßt sich für das vierte Beispiel [HosW83] auf der Ebene des Algorithmus kein Verfahren angeben, mit dem eine schnellere Berechnung über Parallelarbeit zu erreichen ist.

Es lassen sich bei Formulierung der Berechnung von x^{2^k} als Programm jedoch durchaus Formen der Parallelarbeit denken, die einen - wenn auch nicht großen - Geschwindigkeitsgewinn versprechen. Eine Möglichkeit besteht z. B. darin, daß Kontrolloperationen (Prüfung auf Berechnungsende) und Rechenoperationen von verschiedenen Prozessoren ausgeführt werden. Dies zeigt, wie problematisch der Gebrauch des Begriffs "nicht parallelisierbar" ist.

Wie Arbeit aufgeteilt werden kann, wie diese Teile simultan bearbeitet werden können und welche Probleme dabei zu lösen sind, ist Hauptthema der nachfolgenden Kapitel.

Hier soll eine erste Charakteristik des untersuchten Typs von Programmen gegeben werden.

Def. 1.1: Ein ordnungserhaltendes Programm⁷⁾ ist ein Programm, in dem zu verarbeitende Daten in einer vorgeschriebenen Reihenfolge (sequentiell) eingelesen und erzielte Ergebnisse in einer vorgeschriebenen Reihenfolge (sequentiell) ausgegeben werden müssen.

Def. 1.2: Ein ordnungserhaltendes Programmsystem ist eine Zusammenfassung zueinander gehöriger ordnungserhaltender Programme.

Beispiele ordnungserhaltender Programmsysteme sind Compiler und Interpreter oder etwa Programme zur Auswertung zeitlich sequentiell anfallender Meßwerte u. ä.

Beispiele nicht ordnungserhaltender Programmsysteme (im folgenden als ordnungsfreie Programmsysteme bezeichnet) sind alle Programme, die wesentlich auf Datenstrukturen arbeiten, die nicht in einer fest vorgegebenen Reihenfolge "gefüllt" oder "geleert" werden müssen, etwa Programme zur elementweisen Transformation von Vektoren oder Matrizen.

Die Analyse ordnungserhaltender Programmsysteme in bezug auf Parallelisierbarkeit erscheint durch die Vorgaben der Sequentialität schwerer zugänglich als die ordnungsfreier Programmsysteme. Doch führt sie, wie sich zeigen wird, zu Strategien, die für beliebige (auch ordnungsfreie) Programmsysteme einsetzbar sind

7) Eine ordnungserhaltende Funktion f ist für reelle Zahlen als Argument in der Weise definiert, daß aus der Beziehung

$$a < b$$

die Beziehung

$$f(a) < f(b)$$

folgt.

Für ordnungserhaltende Programme gilt eine analoge Beziehung für zu verarbeitende Daten und der Relation "ist zu verarbeiten vor".

und zu Problemen in der parallelen Programmierung, die für "leicht" parallelisierbare Systeme nicht notwendigerweise entstehen, jedoch immer als Möglichkeit einkalkuliert werden müssen. Es läßt sich erwarten, daß gerade die Parallelisierung solcher Systeme einen Beitrag zu einem besseren Verständnis eines methodischen Ansatzes zur Implementierung paralleler Programme und zur Definition einer hierzu geeigneten Programmierungsumgebung leisten kann.

1.2 Hierarchische Multiprozessorsysteme mit eingeschränkten Ressourcen

Die Implementierung von Programmsystemen auf Multiprozessoranlagen wird stark von der verfügbaren Hardware beeinflusst. Wesentliches Kriterium sind Zahl der Prozessorelemente und ihre Verbindung. Ausgehend vom allgemeinen Modell eines Multiprozessorsystems der Abb. 1.1 (vgl. etwa [From82]) gelten in Anlehnung an [Wen78] die folgenden Definitionen:

Def. 1.3: Ein Prozessorelement besteht aus einem Prozessor⁸⁾, dem ggf. ein lokaler Speicher zugeordnet ist.

Def. 1.4: Ein Multiprozessorsystem besteht aus mindestens zwei zusammengeschalteten Prozessorelementen.

8) Hier wie auch im folgenden soll unter einem Prozessor die Zusammenfassung von Programmsteuerung (Leitwerk) und Rechenwerk mit ihren Bestandteilen (Befehlszähler, Akkumulator, Mikroprogramm usw.) im Sinn des klassischen Universal-Rechenautomaten verstanden werden.

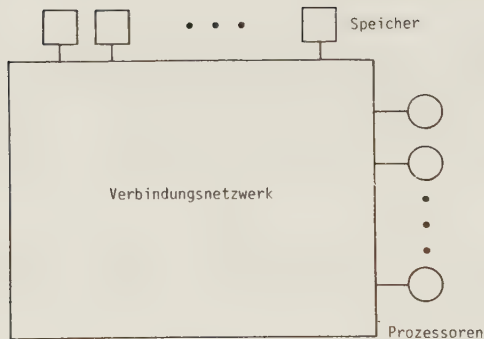


Abb. 1.1: Allgemeines Modell eines Multiprozessorsystems

Def. 1.5: In einem symmetrischen Multiprozessorsystem ist jeder Prozessor in der Lage, alle im Rahmen des Betriebsablaufs anfallenden Aufgaben durchzuführen.

Def. 1.6: In einem asymmetrischen Multiprozessorsystem sind nicht alle Prozessoren zu allen im Rahmen des Betriebsablaufs anfallenden Aufgaben befähigt.

Während die Definitionen 1.5 und 1.6 Eigenschaften einzelner Prozessoren im Multiprozessorsystem beschreiben, bezieht sich der Begriff "Hierarchie" auf die Verbindungsstruktur der Prozessorelemente.

Def. 1.7: Ein Multiprozessorsystem heißt hierarchisch, wenn die Verbindungsstruktur seiner Prozessorelemente hierarchisch ist.
Wenn die Verbindungsstruktur der Prozessorelemente keine Hierarchie erkennen läßt, wird das System als verteilt bezeichnet.

Kennzeichen hierarchischer Multiprozessorsysteme ist eine baumartige Anordnung der einzelnen Prozessorelemente (eine Wurzel; "viele" Blätter, die von der Wurzel über mehrere Knoten hinweg verbunden sind; evtl. weitere Verbindungen zwischen den Knoten).

Die hierarchische Struktur findet sich in der Regel auch in der Gestaltung von Betriebssystem und Anwenderprogrammen wieder (vgl. nachfolgende Kapitel). Der Wurzel werden Kontrollfunktionen, den Blättern Arbeitsfunktionen zugeordnet. Knoten in Zwischenebenen realisieren in erster Linie den Informationsaustausch zwischen Wurzel und Blättern.

Entsprechend werden Betriebssystem und Anwenderprogramme so gestaltet, daß Prozessorelemente höherer Schichten über mehr Fähigkeiten als Prozessorelemente tieferer Schichten verfügen. (Dies sind im Gegensatz zu Def. 1.5 und Def. 1.6 keine Fähigkeiten, die durch die Hardware bedingt sind.)

Zu den hier angesprochenen besonderen Fähigkeiten zählen etwa

1. Erweiterter Zugriff auf Dienste des Betriebssystems
2. Fähigkeiten, Prozesse auf anderen Prozessoren zu starten oder zu stoppen
3. Möglichkeiten, Multiprogramming (d. h. den Ablauf mehrerer Prozesse auf einem Prozessor) zu realisieren

Beispiel eines hierarchischen Multiprozessorsystems ist die den Untersuchungen zugrundeliegende EGPA-Pilotpyramide [HHS76], [HHHS83], deren Struktur in Abb. 1.2 dargestellt ist. Besondere vom Betriebssystem vorgesehene Fähigkeiten des B-Prozessors sind (neben dem Zugriff auf den gesamten Speicher) der Zugang zu Betriebssystemfunktionen, die den A-Prozessoren nicht direkt zugänglich sind (z. B. E/A-Funktionen) bzw. überhaupt nicht möglich sind (z. B. Funktionen zum Starten und Stoppen fremder Prozesse). In [Dre83] wird ein Überblick über die Möglichkeiten der verschiedenen Prozessortypen gegeben, auf Funktionen des Betriebssystems zuzugreifen.

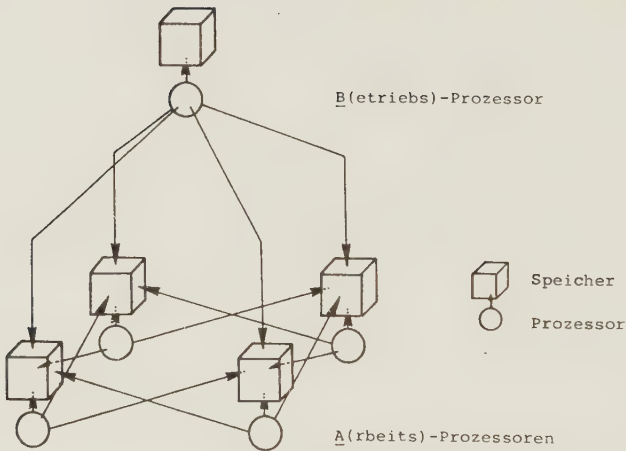


Abb. 1.2: Struktur des EGPA-Systems (EGPA-Pilotpyramide)

Viel stärker als in Monoprozessorsystemen ist der Programmierer in Multiprozessorsystemen mit dem Problem konfrontiert, "hardwareangepaßt" zu programmieren. Die Zahl und die Verbindungsstruktur von Prozessorelementen, die Größe des verfügbaren Speichers, die Zahl und Art möglicher Mechanismen zur Kommunikation sind Größen, die Einfluß auf Programmpartitionen haben können. Die Implementierung eines "eleganten" Verfahrens zur Lösung eines vorgegebenen Problems kann scheitern, wenn in letzterem Sinn verstandene Ressourcen nicht in ausreichender Zahl und Kapazität zur Verfügung stehen.

Wenn an dieser Stelle von einer Einschränkung verfügbarer Ressourcen gesprochen wird (es gibt ja keine realen Multiprozessorsysteme mit uneingeschränkten Ressourcen), ist damit eine Einschränkung gemeint, die entscheidenden Einfluß auf die Wahl eines Verfahrens zur Implementierung von Programmsystemen auf Multiprozessoranlagen hat.

Die Einbeziehung solcher Grenzen führt zu Lösungen, die auch bei Aufhebung solcher Einschränkungen möglich sind, dann aber weniger effizient sein können als Lösungen, die solche Einschränkungen von vornherein ausschließen.

In diesem Sinn soll die "praxisorientierte" Form der Vorgehensweise nicht als Eingrenzung, sondern als Ansporn zur Entwicklung und Bewertung von Verfahren zur Implementierung von Programmsystemen auf Multiprozessorsystemen dienen, die "leicht" an verschiedene Umgebungen anpaßbar sind und die vorhandenen Ressourcen in möglichst "guter" Weise nutzen.

1.3 Eine informelle Definition des Begriffs "parallele Programmierumgebung"

Die Entwicklung von Parallelprogrammen wird neben der Hardwarestruktur des Multiprozessorsystems maßgeblich von der Güte der vorhandenen Programmierhilfsmittel und dabei insbesondere von der Benutzerschnittstelle zum Betriebssystem beeinflusst. Dabei ergibt sich ein Dilemma zwischen dem Maß an wünschenswertem Komfort, der möglichen Flexibilität und der erreichbaren Effizienz. Komfortable Schnittstellen können Effizienzeinbußen zur Folge haben und die Strategien zur Erzielung zufriedenstellender Lösungen zu stark einschränken. Die Verwendung elementarer Programmierhilfsmittel führt zwar zu effizienten Lösungen, unterstützt jedoch den Benutzer in der Regel nicht bei Programmentwicklung und Fehlersuche und verlangt im allgemeinen die Kenntnis maschinen- und systemspezifischer Details.

Die parallele Programmierumgebung stellt ein Bindeglied zwischen Benutzerprogramm bzw. Programmiersprache und Schnittstelle zum Betriebssystem bzw. zur Hardware dar. Sie verdeckt Details von Betriebssystem und Hardware gegenüber dem Anwender und stellt Hilfen zur Entwicklung und zum Test von Parallelprogrammen bereit.

Der Umfang an durch die parallele Programmierumgebung bereitgestellten Hilfen und die Art des Zugangs ist für die Entwicklung von Parallelprogrammen von großer Bedeutung. Die Frage nach Kriterien einer "guten" parallelen Programmierumgebung kann nicht losgelöst vom System und den implementierten Anwendungen gesehen werden. Deshalb ist ein weiteres Ziel, aus der Implementierung paralleler Programmbeispiele mit verschiedenen Programmierhilfsmitteln heraus, zu Aussagen zu gelangen, in welcher Form und über welche Schnittstellen eine Unterstützung der Parallelprogrammierung durch eine geeignet definierte parallele Programmierumgebung erfolgen kann und in welcher Form diese für verschiedene Systeme definiert werden kann.

Die experimentelle Form der Vorgehensweise erlaubt eine qualitative und quantitative Bewertung der vorgestellten Strategien und Schnittstellen.

2 Vom sequentiellen Programm zu parallelen Verarbeitungsstrategien

Als Grundlage zur Erläuterung von Strategien zur Bearbeitung ordnungserhaltender Programmsysteme auf Multiprozessorsystemen wird in diesem Kapitel eine Systematik zum Übergang von einem sequentiellen Programm zu einem sequentiell partitionierten Programm beschrieben. Das sequentiell partitionierte Programm ist nicht nur Ausgangspunkt paralleler Versionen, sondern kann in seiner Struktur auch als sequentielle Testumgebung für parallele Programmfassungen angesehen werden.

Durchgeführte Verarbeitungsschritte werden soweit möglich an einem einfachen Programmbeispiel (Sortierung einer Zahlenfolge) verdeutlicht.

Kap. 2.1 gibt eine Zusammenstellung von aus der Literatur bekannten Überlegungen und Methoden, welche als Ausgangspunkt zur Erläuterung der in den Kapiteln 2.2. und 2.3 durchgeführten Umformungen des sequentiellen Ausgangsprogramms in eine Menge von (parallel ausführbaren) Teilaufgaben dienen.

2.1 Überlegungen und Methoden zur Parallelisierung von Programmen

Der Ausgangspunkt jeglicher Parallelisierung besteht darin, eine gegebene Aufgabe in Teilaufgaben zu zerlegen, die dann von mehreren Verarbeitungseinheiten (z. B. Prozessoren) gleichzeitig ausgeführt werden, wobei während der Ausführung eine Kommunikation zwischen den verschiedenen Teilen der Gesamtaufgabe stattfindet.

Nach [Kung80] lassen sich aus den verschiedenen Verfahren entstandene parallele Algorithmen nach folgenden Kriterien klassifizieren:

1. Wie die Kontrolle über den parallelen Ablauf ausgeübt wird (zentral, dezentral, synchron, asynchron, ...); vgl. hierzu Kap. 3
2. wie groß die einzelnen Teilaufgaben sind und damit, wie häufig Kommunikationsvorgänge erforderlich werden (die sog. Modulgranularität)
3. wie die Kommunikationsgeometrie im Algorithmus beschaffen ist (wie die verschiedenen Teilaufgaben untereinander vernetzt sind)

Nachfolgend werden zunächst Bedingungen und Überlegungen zur parallelen Ausführbarkeit von Teilen einer Aufgabe, anschließend einige darauf aufbauende elementare Verfahren für ideale Rechensysteme zusammengestellt, die Grundlagen der nachfolgenden Abschnitte zur Beschreibung des Ansatzes zur Parallelisierung von "großen" Aufgaben für reale Multiprozessorsysteme darstellen.

2.1.1 Zur parallelen Ausführbarkeit von Aufgaben

Grundlage dafür, daß zwei Teilaufgaben eines Programms parallel⁹⁾ ausgeführt werden können, ist, daß die betreffenden Teilaufgaben sequentiell in einer beliebigen Reihenfolge ausgeführt werden können, ohne daß sich am Programmresultat etwas ändert (Determiniertheit).

Zwei Teilaufgaben sind dann nicht parallel zueinander ausführbar, wenn das Ergebnis der einen Teilaufgabe vom Ergebnis der anderen Teilaufgabe abhängig ist. Abhängigkeiten zwischen Teilaufgaben eines Programms existieren auf der Ebene des Kontrollflusses

9) Auf verschiedene Formen einer parallelen Ausführung von Teilaufgaben (pipelineartig, nebenläufig, gemischt pipelineartig/nebenläufig) wird in Kap. 3 eingegangen.

(z. B. eine Programmverzweigung in Abhängigkeit des Wertes einer durch eine andere Teilaufgabe veränderten Größe) und des Datenflusses (z. B. Verwenden des Resultates einer Teilaufgabe als Eingabe einer anderen Teilaufgabe).¹⁰⁾

Die Ebene des Kontrollflusses läßt sich mit der Ebene des Datenflusses verschmelzen, wenn die den Kontrollfluß eines Programmes bestimmenden Größen als Eingaben der entsprechenden Teilaufgabe interpretiert werden und Ergebnisse von Tests, die zu Programmverzweigungen führen, in einer Art Pseudovariablen als Eingabe für nachfolgende Operationen hinterlegt werden. Dann lassen sich nach [Baer73] und [Bern66] notwendige Bedingungen angeben, die erfüllt sein müssen, um die parallele Ausführbarkeit zweier Teilaufgaben zu ermöglichen. Sie lassen sich in folgendem Satz zusammenfassen:

Satz 2.1: Seien T_i und T_j ($i \neq j$) zwei Teilaufgaben eines Programms; sei mit E_i und E_j die Menge der Eingabevariablen und mit A_i und A_j die Menge der Ausgabevariablen von T_i und T_j bezeichnet. Dann ist folgende Bedingung notwendig, damit T_i und T_j parallel ausgeführt werden können:

$$E_i \cap A_j = E_j \cap A_i = A_i \cap A_j = \emptyset (*)$$

Damit ist gesagt, daß zwei Teilaufgaben sicher dann nicht parallel ausgeführt werden können, wenn die eine Teilaufgabe Ergebnisse als Eingaben benötigt, die von der anderen Teilaufgabe geliefert werden oder wenn die beiden Teilaufgaben zur Ablage ihrer Ausgaben zum Teil gleiche Speicherbereiche verwenden.

¹⁰⁾ Bei den hier angestellten Überlegungen zur parallelen Ausführbarkeit von Teilaufgaben soll davon ausgegangen werden, daß der mögliche Parallelismus nicht durch die Menge der verfügbaren Verarbeitungseinheiten begrenzt sein soll.

Eine äquivalente Formulierung notwendiger Bedingungen, die die parallele Ausführbarkeit von Teilaufgaben verhindern, ist auf der Ebene des Befehlswerks der CDC 6600 durch die Konflikte erster, zweiter und dritter Art gegeben (vgl. hierzu etwa [BoHä83]).

Wie in [Bern66] gezeigt wird, ist die parallele Ausführbarkeit zweier Teilaufgaben T_i und T_j allgemein nicht entscheidbar. Dies liegt darin begründet, daß Programme konstruierbar sind, in denen die parallele Ausführbarkeit von Teilaufgaben von einer nicht entscheidbaren Bedingung (z. B. ob eine Turingmaschine bei einer bestimmten Eingabekonfiguration nach endlicher Zeit oder nie zum Halten kommt) abhängt.

Probleme der Unentscheidbarkeit bezgl. der parallelen Ausführbarkeit treten nicht auf, wenn von Programmen ausgegangen wird, die nach (endlich) bestimmbarer Zeit terminieren. Dann läßt sich das Programm als schleifenfreier gerichteter Graph formulieren. Eine in solchen Programmen leicht durchführbare Konstruktion, die eine automatisierbare Erkennung von Parallelismus ermöglicht, ist die Realisierung der Single-Assignment-Regel, d. h. dafür zu sorgen, daß im Programmlauf an jede Variable höchstens einmal ein Wert zugewiesen wird.

In Programmen bzw. Sprachen, in denen die Single-Assignment-Regel per Definition festgeschrieben wird (ein Verstoß wird durch den Compiler angezeigt), den sog. Single-Assignment-Sprachen (z. B. [Com78]), ist dann auf einfache Weise entscheidbar, wann eine Operation frühestens (auch parallel zu anderen) ausgeführt werden kann, nämlich dann, wenn an jede der von ihr benötigten Eingabevariablen (Operanden) ein Wert zugewiesen wurde. Nachdem einem Operanden ein Wert zugewiesen wurde, wird dieser als "verfügbar" bezeichnet.

Auf mit Single-Assignment-Sprachen zusammenhängende Probleme (z. B. eine neue Form zur Beschreibung von Kontrollstrukturen wie Loops) soll an dieser Stelle nicht weiter eingegangen werden. Hierzu sei etwa auf [Com78] verwiesen.

Der Einbau eines reinen Single-Assignment-Konzeptes in real existierende Programmiersprachen und Rechensysteme stößt wegen der immensen Zahl der hierzu benötigten Variablen auf Schwierigkeiten. Deshalb werden zur Realisierung Überlegungen notwendig, wie diese Zahl ohne Einbuße an möglicher Parallelität durch Mehrfachnutzung von Variablen reduziert werden kann. Dazu sind dann Betrachtungen über den "Lebenszeitraum" einer Variablen notwendig, d. h. die Feststellung des Zeitpunktes, zu dem zum ersten Mal auf die entsprechende Variable bzw. des Zeitpunktes, zu dem zum letzten Mal auf die entsprechende Variable zugegriffen wird. In [Rou77] wird eine Methode beschrieben, über solche Betrachtungen den Bedarf an Variablen in Single-Assignment-Programmen zu minimieren.

Wesentlich erscheint, daß es die Single-Assignment-Eigenschaft erlaubt, effektiv durch Auswertung der Eingabe- und Ausgabemengen von Operationen unter Berücksichtigung der Bedingung (*) des Satzes 2.1 ein maximal paralleles Ablaufschema zu bestimmen, welches in graphischer Form als schleifenfreier Präzedenzgraph bzw. in algebraischer Form als Konnektivitätsmatrix dargestellt werden kann [Rama69]. Abb. 2.1 zeigt hierzu ein Beispiel.

Wie aus Abb. 2.1 ersichtlich, führen die Überlegungen zur Beschreibung der Abhängigkeit von Teilaufgaben im allgemeinen auf nicht zusammenhängende gerichtete Graphen. Diese lassen sich zu einem zusammenhängenden Graphen verschmelzen, wenn man alle Knoten ohne einlaufende Kanten mit einem "Initial"-Knoten und alle Knoten ohne auslaufende Kanten mit einem "Final"-Knoten verbindet. Auf diese Weise erhält man bezgl. der Teilaufgaben eines Programms einen Verband, der durch die Relation "ist auszuführen vor" geordnet ist. In diesem Verband können alle Teilaufgaben parallel zueinander ausgeführt werden, die bezgl. der Relation "ist auszuführen vor" in keiner Beziehung zueinander stehen.

Teil- auf- gaben	Va- riab- le								
	v ₁	v ₂	v ₃	v ₄	v ₅	v ₆	v ₇	v ₈	v ₉
t ₁									
t ₂									
t ₃									
t ₄	1								
t ₅				1					
t ₆	1								
t ₇									
t ₈		1							
t ₉		1	1				1		

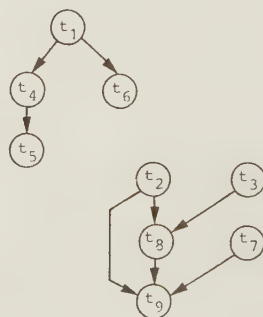
Eingabematrix enthält 1 auf Posi-
tion $[i,j]$, wenn v_j Eingabe von t_i

	v ₁	v ₂	v ₃	v ₄	v ₅	v ₆	v ₇	v ₈	v ₉
t ₁	1								
t ₂		1							
t ₃			1						
t ₄				1					
t ₅					1				
t ₆						1			
t ₇							1		
t ₈								1	
t ₉									1

Ausgabematrix enthält 1 auf Posi-
tion $[i,j]$, wenn v_j Ausgabe von t_i

	t ₁	t ₂	t ₃	t ₄	t ₅	t ₆	t ₇	t ₈	t ₉
t ₁				1	1	1			
t ₂							1	1	
t ₃							1	1	
t ₄					1				
t ₅									
t ₆									
t ₇								1	
t ₈								1	
t ₉									

Konnektivitätsmatrix enthält auf Posi-
tion $[i,j]$ die 1, wenn t_i vor t_j auszu-
führen ist.



Zugehöriger
Präzedenzgraph

Abb. 2.1: Beispiel zur Beschreibung eines maximal
parallelen Ablaufschemas für ein Single-
Assignment-Programm

Ein dem Single-Assignment-Konzept äquivalenter Ansatz zur "automatischen" Erzeugung parallel ablauffähiger Teilaufgaben wird in (reinen) Datenflußkonzepten verfolgt. Auch hier wird der Ausführungszeitpunkt einer Operation aufgrund der Verfügbarkeit bzw. Nicht-Verfügbarkeit von Operanden bestimmt. Mit Ausführung der Operation werden die Eingaben jedoch im Gegensatz zur Philosophie der Single-Assignment-Sprachen "verbraucht" (indem sie durch die erzeugten Ergebnisse ersetzt werden), sodaß diese Eingaben anschließend nicht mehr für andere Operationen zur Verfügung stehen.

In Datenflußmechanismen des Typs "data-driven" wird eine Operation dann ausgeführt, wenn die zugehörigen Eingabeoperanden verfügbar sind. In Mechanismen des Typs "demand-driven" wird eine Operation dann ausgeführt, wenn ihr Ergebnis als Eingabeoperand einer anderen Operation benötigt wird. In [Back78] wird eine Sprache zur Realisierung des Typs "demand-driven" beschrieben.

Eine Beschreibung der Methoden des Datenflußkonzepts findet sich etwa in [Trel79], zu Beispielen von Datenflußsprachen und Kriterien ihrer Realisierung sei auf [Acke79] verwiesen. [Denn80] behandelt Fragen der Architektur von Datenflußrechnern, in [TrBH82] wird eine Zusammenstellung existierender Datenflußrechner gegeben und die Datenflußmechanismen dem in üblichen Programmiersprachen nachgebildeten Kontrollfluß-Mechanismus gegenüberstellt.

Wenn auch die hier skizzierten Methoden zur Bestimmung maximal paralleler Abarbeitungsreihenfolgen nicht allgemein für die Parallelisierung großer Programmsysteme auf der Ebene von Anweisungen einsetzbar sind (die benötigten Matrizen würden viel zu viel Speicher benötigen bzw. das eigentliche Verfahren viel zu viel Zeit verbrauchen), so bilden sie doch eine wichtige Grundlage zur Parallelisierung "kleiner" Programme (Kap. 2.1.2) bzw. zur Parallelisierung "großer" Programme auf der Ebene größerer logischer Programmeinheiten, in denen Single-Assignment- und Datenflußkonzepte in einer höheren Ebene nachmodelliert werden (Kap. 2.2).

Es ist sicher ein Resultat der unbefriedigenden Situation bei der automatischen Parallelisierung, daß im Verlauf der Entwicklung paralleler Programmstrukturen eine Fülle von Rechnerarchitekturen entstand, die in der Lage sind, bestimmte Typen von Aufgaben unter Verwendung vieler Verarbeitungseinheiten besonders "gut" parallel zu lösen. Beispiele hierfür sind neben den bereits erwähnten Datenflußarchitekturen Vektorrechner, Feldrechner oder auch Rechner, die eine pipelineartige Ausführung von Teilaufgaben auf der Ebene von Mikroinstruktionen, Rechenwerk, Befehlswerk oder E/A-Einheiten ermöglichen (vgl. Kap. 3.1). Ausgangspunkt nachfolgend beschriebener Untersuchungen sind jedoch Systeme, die aus "allgemeinen", für sich programmierbaren Prozessorelementen bestehen. In [BoHä83] wird eine umfassende Übersicht über Prinzipien des Parallelismus und daraus entstandenen Beispielen von parallelen Rechnerarchitekturen gegeben.

2.1.2 Verfahren in idealen Rechensystemen

Allgemeine von einem speziellen Rechnersystem losgelöste Parallelisierungsverfahren gehen in der Regel von einem idealen Rechnersystem aus, wie es etwa in [HosW83] zusammengefaßt ist:

1. Das System besteht aus p individuellen Prozessoren, wobei je nach Situation von einem fest vorgegebenen oder beliebig großen Wert p ausgegangen wird.
2. Die zu behandelnden Probleme lassen sich darstellen als parallel oder sequentiell zu verarbeitende Menge von zweistelligen arithmetischen oder logischen Operationen.
3. Jeder einzelne der p Prozessoren kann zu jedem Zeitpunkt eine dieser Operationen ausführen.
4. Verschiedene Prozessoren des Systems können zu jedem Zeitpunkt verschiedene Operationen ausführen.
5. Für die Ausführung einer Operation wird genau eine Zeiteinheit benötigt.
6. Andere als die genannten Operationen (z. B. Datentransfer,

Prozessorsteuerung) sollen keine Zeit beanspruchen.

7. Zugriffskonflikte auf Speichereinheiten und Prozessoren sollen nicht auftreten.

Diese Annahmen erlauben es, zu einer Problemdimension n von parallelen Algorithmen ein abstraktes Komplexitätsmaß $O(n)$ zu definieren. Mit der Bezeichnung $O(n)$ für die Komplexität eines Algorithmus wird ausgedrückt, daß die Zeit zur Ermittlung der Ergebnisse in linearem Zusammenhang mit der Problemgröße n steht. In idealen Multiprozessorsystemen kann eine parallele Fassung nicht mehr als n -mal so schnell wie mit einem Monoprozessor ausgeführt werden. Zur Erreichung dieses Grenzwertes sind mindestens n Prozessoren erforderlich.

Ein für hierarchische Multiprozessorsysteme direkt anwendbares Verfahren zur Parallelisierung, welches auf diesen Annahmen aufbaut, ist das der sog. Baumhöhenreduktion bei arithmetischen Ausdrücken (vgl. etwa [Baer73], [HosW83], [Kuck77]). Abb. 2.2 zeigt beispielsweise, daß die Bildung einer Summe aus acht Elementen durch rekursives Doppeln mit vier Prozessoren in drei Schritten, d. h. $\frac{7}{3}$ mal so schnell wie mit einem Prozessor durchgeführt werden kann.

Meist läßt sich diese Form der Parallelisierung innerhalb arithmetischer Ausdrücke jedoch nur anwenden, wenn zuvor der Ausdruck mit Hilfe von Kommutativ-, Assoziativ- und/oder Distributivgesetz umgeformt worden ist, sodaß der tatsächliche Geschwindigkeitsgewinn niedriger ist.

Außerdem ergibt sich keine Geschwindigkeitssteigerung, wenn statt acht nur sieben, sechs oder fünf Operanden addiert werden sollen. Für diesen Fall wird der mögliche Geschwindigkeitsgewinn durch den ganzzahligen Anteil des Zweierlogarithmus bestimmt, weshalb das Verfahren auch als "Logarithmierung" bezeichnet wird.

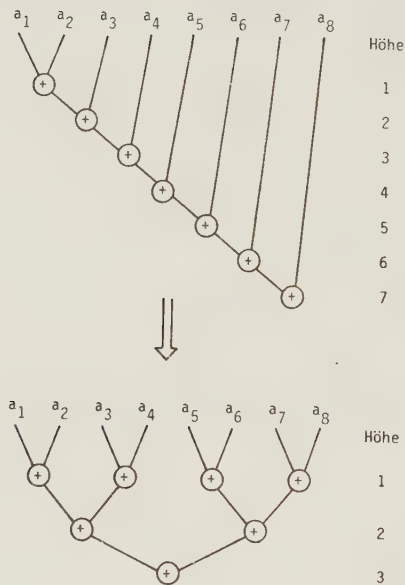


Abb. 2.2: Beispiel zur Baumhöhenreduktion bei der Berechnung einer Summe durch Prozessoren eines Multiprozessorsystems

Auch wenn das auf elementarer Ebene angewendete Verfahren der Baumhöhenreduktion zur Parallelisierung keine dem Hardwareaufwand adäquate Geschwindigkeitssteigerung gegenüber einem Monobetrieb bringt, hat es doch eine Bedeutung bei der Zusammenfassung von Ergebnissen komplexer Berechnungen, die von verschiedenen Prozessoren zusammengetragen werden müssen (vgl. etwa Programmbeispiel des Kap. 4.2).

Verfahren der Parallelisierung über Baumhöhenreduktion lassen sich im allgemeinen nicht direkt in Programmen anwenden, da in praktischen Fällen Programme nicht Anweisungen mit beispielsweise acht oder mehr zu verknüpfenden Operanden enthalten. Ausdrücke dieser Art sind in der Regel in Laufschleifen formuliert. Als Beispiel hierfür sei etwa die folgende Laufschleife zur Berechnung einer Summe von n Elementen angegeben, die in einem Feld der Dimension n stehen:

```
SUMME := 0;
FOR I FROM 1 TO N
DO SUMME := SUMME + FELD[I]
OD;
```

Diese Laufschleife läßt sich äquivalent als arithmetischer Ausdruck der Form

$$\text{SUMME} := \text{FELD}[1] + \text{FELD}[2] + \dots + \text{FELD}[N]$$

schreiben, auf den die Baumhöhenreduktion anwendbar ist.

Mit der Untersuchung von Parallelität innerhalb von Laufschleifen beschäftigen sich eine Reihe von Arbeiten, z. B. [BCKT79], [KMC72], [Lam74].

Hier sollen nur einige typische Beispiele parallelisierbarer und nicht parallelisierbarer Laufschleifen gegeben werden, die auch für die nachfolgenden Verfahren eine gewisse Rolle spielen.

Sehr einfach ist die Parallelisierung von Laufschleifen der Form

```
FOR I TO N DO A[I] := F(B[I], C[I], ...) OD; ,
```

bei denen die Anweisungen eines Schleifendurchlaufs unabhängig von den Anweisungen eines anderen Schleifendurchlaufs und damit parallel zueinander berechnet werden können. (Laufschleifen dieser Art treten etwa bei der elementweisen Transformation von Vektoren oder Matrizen auf.)

Ein etwas weniger "leicht" parallelisierbares Beispiel einer Laufschleife ist die Form

```

FOR L TO N
DO FOR I FROM 2 TO N-1
  DO FOR J FROM 2 TO N-1
    DO U[I,J] := 0.25 * (U[I-1,J] + U[I,J-1]
                        + U[I+1,J] + U[I,J+1])
  OD
OD
OD; ,

```

wie sie bei der numerischen iterativen Auswertung der Differentialgleichung

$$u_{xx} + u_{yy} = 0$$

entsteht.

Ein Verfahren zur Parallelisierung dieser Form ist als sog. Wellenfront-Methode bekannt (vgl. [KMC72], [Kuck77], [Lam74]). Diese ermöglicht die parallele Ausführung von Rechnungen in Laufschleifen dadurch, daß die Reihenfolge, in der einzelne Ergebnisse innerhalb des Rumpfes der Laufschleife berechnet werden, permutierbar ist, ohne daß sich am Gesamtergebnis etwas ändert. Innerhalb der permutierten Reihenfolge ist es dann möglich, mehrere Rechnungen gleichzeitig durchzuführen.

Im vorliegenden Beispiel macht man sich leicht klar, daß innerhalb einer Iteration nach Berechnung eines $U[I,J]$ eine gleichzeitige Berechnung der beiden Größen $U[I+1,J]$ und $U[I,J+1]$ möglich ist. So können z. B. nach der Berechnung von $U[2,2]$ die Größen $U[2,3]$ und $U[3,2]$, danach die Größen $U[2,4]$, $U[3,3]$ und $U[4,2]$ parallel zueinander berechnet werden. Damit ist der sequentielle Algorithmus in einen parallelen Algorithmus übergeführt, der gemäß der Klassifikation von [Kung80] an der Grenze zwischen systolischen und SIMD-Algorithmen eingeordnet werden kann.

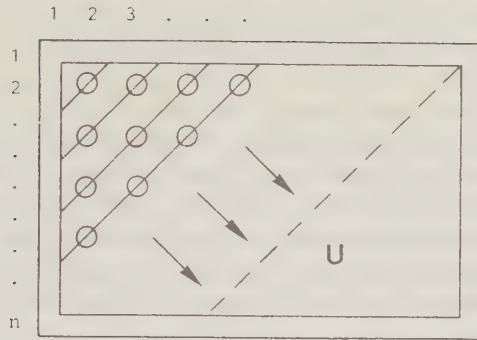


Abb. 2.3: Wellenfront-Methode für die Ausführungsreihenfolge von Anweisungen in Laufschleifen

In Abb. 2.3 ist die Berechnungsrichtung als Wellenfront dargestellt, die von U[2,2] im Winkel von 45° ausgeht.

Eine verallgemeinernde algorithmische Formulierung des Verfahrens ist in [Lam74] zu finden. Hier ist von Bedeutung, daß es möglich ist, durch Permutieren von Anweisungen innerhalb eines Programms eine zunächst nicht sichtbare Parallelität in der Abarbeitung sichtbar zu machen.

Grenzen bei der Parallelisierung von Durchläufen von Laufschleifen ergeben sich schnell dann, wenn der Rumpf aus GOTO-Anweisungen, IF-Anweisungen oder Unterprogrammaufrufen besteht.

So ist es mit keinem algorithmischen Verfahren möglich, den Parallelismus zu bestimmen, der in Laufschleifen der Form

```
FOR I TO N DO IF A[I] = 0 THEN FIRSTNULL := I; BREAK FI OD;
```

oder

```
FOR I TO N DO A[I] := A[F(I)] OD;
```

enthalten ist.

Es lassen sich zwar einige Verfahren zur Formulierung parallel ausführbarer Teilaufgaben über die Parallelisierung "einfach" gebauter Laufs Schleifen geben. Laufs Schleifen sind generell auch einer der wichtigsten Ansatzpunkte zur Formulierung paralleler Verarbeitungsstrategien. Es läßt sich jedoch kein algorithmisches Verfahren angeben, welches eine Parallelisierung beliebiger Arten von Laufs Schleifen zu beschreiben gestattet.

Hier zusammengestellte Verfahren, die eine Parallelisierung von Programmen auf der Grundlage einzelner Anweisungen vornehmen, werden in realen Multiprozessorsystemen der betrachteten Art im allgemeinen aus Effizienzüberlegungen nicht eingesetzt.

Sie bieten jedoch, wie die weiteren Ausführungen zeigen, eine Grundlage für Parallelisierungsstrategien, die auf größeren Programmeinheiten aufsetzen.

2.2 Zur Bildung von Teilaufgaben für Prozessoren eines Multiprozessorsystems, die E/A-Schnittstelle

Konzepte und Sprachentwürfe für die Implementierung von Programmen auf Multiprozessorsystemen definieren in der Regel Prozesse oder Tasks, um Parallelität auszudrücken.

Def. 2.2: Prozesse heißen parallel, wenn sich ihre zeitliche Ausführung überlappt, d. h. wenn die erste Aktion eines Prozesses beginnt, bevor die letzte Aktion eines anderen bereits ablaufenden Prozesses endet.

In nachfolgenden Kapiteln werden vornehmlich zwei Klassen von parallelen Prozessen betrachtet, nebenläufige Prozesse und pipelineartig ablaufende Prozesse. Diese werden wie folgt definiert.

Def. 2.3: Parallele Prozesse heißen nebenläufig, wenn sie sich nicht in ihrem Ablauf beeinflussen.

Def. 2.4: Zwei parallele Prozesse laufen pipelineartig ab, wenn im Ablauf der beiden Prozesse zu bestimmten Zeitpunkten auf einem Prozeß Ergebnisse entstehen, die vom anderen Prozeß zur Weiterarbeit benötigt werden.

Eine Menge paralleler Prozesse läuft pipelineartig ab, wenn jeder Prozeß dieser Menge zu mindestens einem anderen Prozeß dieser Menge pipelineartig abläuft.

Pipelineartige Parallelität ist eine Form von abhängiger Parallelität, die durch den Kontroll- und Datenfluß zwischen den beteiligten Prozessen entsteht. Pipelineartig ablaufende Prozesse kooperieren miteinander und warten gegebenenfalls aufeinander.

Eine andere Form von abhängiger Parallelität entsteht, wenn Prozesse um das gleiche nur exklusiv benutzbare Betriebsmittel konkurrieren. Dann entstehen Abhängigkeiten zwischen parallelen Prozessen, wenn ein Prozeß auf die Freigabe eines solchen Betriebsmittels durch einen anderen Prozeß wartet. In Kap. 5.2 wird auf einige Schlußfolgerungen dieser (eigentlich ungewollten) Abhängigkeit zwischen parallelen Prozessen für die Implementierung von Parallelprogrammen eingegangen. Für den Begriff "Parallelprogramm" soll dabei wie auch im weiteren von folgender Definition ausgegangen werden.

Def. 2.5: Ein Parallelprogramm besteht aus einer Menge miteinander kooperierender paralleler Prozesse.

Die Definition verlangt nicht, daß jeder Prozeß dieser Menge mit jedem anderen Prozeß der Menge direkt kooperiert. Es wird aber vorausgesetzt, daß jeder Prozeß eines Parallelprogramms mit jedem anderen Prozeß des Parallelprogramms - möglicherweise über mehrere Prozesse hinweg - kooperieren kann.

Die Zusammenarbeit zwischen den parallelen Prozessen eines Parallelprogramms erfolgt dadurch, daß die einzelnen Prozesse untereinander Nachrichten austauschen. Der Nachrichtenaustausch zwischen verschiedenen Prozessen erfolgt über Zugriff auf Variable in einem gemeinsam benutzten Speicherbereich oder über Botschaften. Eng mit der Art des Nachrichtenaustauschs verknüpft ist die Frage, welcher Synchronisationsmechanismus im Prozeßsystem benutzt wird.

In [Win79] und [Win80] wird eine umfassende Zusammenstellung gegeben, wie das Prozeßkonzept in Betriebssystemen und Programmiersprachen realisiert ist. Eine allgemeine Zusammenfassung von Konzepten und Notationen für die Parallelprogrammierung findet sich in [AnSch83]. Auf Fragen dieser Art wird in Kap. 5 eingegangen.

Methoden des Kapitels 2.1 lassen sich formal einfach auf derart beschriebene Systeme übertragen, wenn jede Anweisung als eigener Prozeß definiert wird.

Eine direkte Umsetzung idealer in real existierende Multiprozessorsysteme ist jedoch nicht sinnvoll. Abgesehen von der eingeschränkten Zahl zur Verfügung stehender Prozessoren gelten insbesondere die Annahmen der Zeitlosigkeit organisatorischer Operationen wie Datentransfer und Prozessorsteuerung nicht.

Der Einfluß organisatorischer Operationen läßt sich nur dann weitgehend zurückdrängen, wenn diese sehr selten auftreten und eine gegenüber Rechenoperationen vernachlässigbare Laufzeit besitzen. In diesem Fall muß die Ebene einzelner Anweisungen zur Aufteilung eines Programms in Teilaufgaben (wie in den Verfahren des Kap. 2.1.2) verlassen werden und als Grundlage von Teilaufgaben Zusammenfassungen von Anweisungen ausgewählt werden.

Eine in Höheren Programmiersprachen als natürlich angesehene Ebene zur Auftrennung von Programmcode ist die der Programmblöcke, der Laufschleifen, der Prozeduren oder der Module.

Die Zerlegung eines umfangreichen Programmsystems in Module hat hauptsächlich den Grund, den Aufwand für Änderungen und Erweiterungen des Programmsystems dadurch klein zu halten, daß in derartigen Fällen nur kleine Einheiten zu betrachten sind. Als ein weiterer sehr gewichtiger Gesichtspunkt einer Modularisierung wird angesehen, daß die Programmentwicklungszeit dadurch verkürzt werden kann, daß verschiedene Personen an Entwurf und Implementierung verschiedener Module gleichzeitig (parallel!) arbeiten können. Dies ist exakt das gleiche Argument, welches für den Einsatz von Multiprozessorsystemen bezgl. der Laufzeit von Programmen Anwendung findet.

Mit Richtlinien und Methoden zur Modularisierung, sowie mit Klassifizierungen von aus verschiedenen Verfahren entstandenen Programm-Modulen beschäftigen z. B. [Alt78], [Den79], [GoKa78], [Par72]. Das wesentliche Ergebnis aller Untersuchungen ist, daß die Methoden im Grunde informell sind. Eine Aufteilung eines Programmsystems in Module kann nicht formal schematisch wie eine Aufteilung an Anweisungsgrenzen erfolgen, sondern muß Aspekte der Syntax und Semantik berücksichtigen (Aufteilung an der Grenze von Programmblöcken; Aufteilung an Stellen, an denen Daten von einer Struktur in eine andere Struktur transformiert worden sind). Subjektiv unterschiedliche Vorstellungen von Inhalten und Strukturen eines Programmsystems zwischen verschiedenen Programmentwicklern können zu verschiedenen "guten" Modularisierungen führen. Dies ist wohl auch der Grund, der noch einer Automatisie-

rung der Parallelisierung auf der Basis von Programm-Modulen entgegensteht.

Eine in ordnungserhaltenden Programmsystemen wie z. B. Compilern im allgemeinen realisierte Form einer Modularisierung folgt den Konstruktionsrichtlinien des Composite Design [My75]. Danach wird die Ausführung eines Programms als ständige Transformation eines Stroms von Eingaben in Ausgaben angesehen. Startpunkt ist die Programmaufteilung in die drei grundlegenden Elementaraufgaben Eingabe, Verarbeitung (Umformung) und Ausgabe. Dann erfolgt eine schrittweise Verfeinerung des Teils Verarbeitung in die Bestandteile Noch-Eingabe, Umformung, Schon-Ausgabe. Dabei sind dann Ausgaben eines Moduls Eingaben für Folgemodule bzw. Eingaben von Modulen Ausgaben früher abgelaufener Module.

Die nach diesem Verfahren entstandenen Verbindungsstrukturen von Modulen lassen sich dann graphisch durch Präzedenzgraphen (Kap. 2.1.1), Netzwerke [Iss83] oder Algorithmenschemata [Klei82] darstellen. Unterschiede zwischen den verschiedenen Schemata bestehen im wesentlichen in der Interpretation der zwischen den einzelnen Modulen fließenden Daten (als Menge, als komplexe Datenstruktur oder als Botschaft).

Die so entstandenen Graphen (Abb. 2.4 zeigt die uninterpretierten Graphen für die Programmbeispiele des Kap. 4) haben eine wesentlich einfachere Struktur, als die, die bei einer Aufteilung des Programms an Grenzen von Anweisungen entstehen würden (soweit die Konstruktion von Graphen auf dieser Grundlage aus Aufwandsgründen überhaupt möglich ist). Sie haben bezgl. Überlegungen zur Parallelisierung aber den Nachteil, daß aus ihnen parallele Abarbeitungsreihenfolgen bezgl. der Ausführung von Modulen nur sehr eingeschränkt bzw. im Fall einer Kette voneinander abhängiger Moduln überhaupt nicht mehr ersichtlich sind.

Die Form der Darstellung impliziert, daß ein Modul erst dann mit der Bearbeitung seiner Aufgabe beginnen kann, wenn sämtliche von ihm zu verarbeitende Eingaben vorliegen und daß sämtliche Ausgaben eines Moduls auf einmal entstehen.

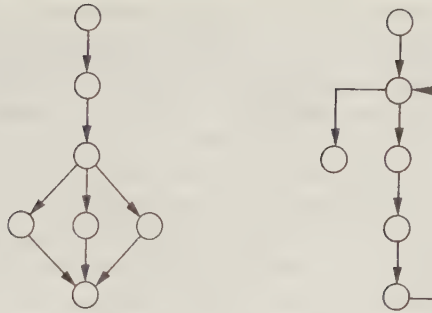


Abb. 2.4: Graphstruktur der Programmbeispiele des Kap. 4 (uninterpretiert)

Dies trifft jedoch für ordnungserhaltende Programmsysteme zumindest für die ursprünglichen Eingaben und die endgültigen Ausgaben nicht zu. Ursprüngliche Eingaben werden portionsweise (satzweise, elementweise) angefordert, abschließende Ausgaben entstehen ebenso in Portionen. Dies gilt im allgemeinen auch für die Aufgaben der dazwischen liegenden Module.

Für die Parallelverarbeitung ist dieses Prinzip der Ströme aus von Zeit zu Zeit entstehenden Daten von Bedeutung, da es die Möglichkeit eines pipelineartigen Arbeitens ermöglicht (näheres hierzu in Kap. 3.1). Das Prinzip der Datenströme hat auch Eingang in Datenflusssprachen gefunden [Acke79].

Nun wird allgemein zur Vermeidung von Ein- und Ausgaben die Übergabe von Ergebnissen eines Moduls an nachfolgende Module in Form von gemeinsam zugreifbaren Daten (z. B. als Parameter von Prozeduren oder in Form globaler Daten) erfolgen. Eine solche Übergabeform der Ergebnisse zwischen verschiedenen Modulen ist innerhalb eines Multiprozessorsystems, welches Kommunikation zwischen verschiedenen Prozessoren allein auf der Basis von Botschaften realisiert, jedoch nur dann möglich, wenn aufeinanderfolgende Module dem gleichen Prozessor zugeordnet sind.

Diese Einschränkung gilt dann nicht, wenn der Austausch von Ergebnissen zwischen verschiedenen Modulen ausschließlich als Senden und Empfangen von Botschaften formuliert wird (vgl. jedoch Fußnote 6 in Kap. 1). Dieser Weg wird in [Iss83] beschritten. Da die meisten Programmiersprachen jedoch kein Botschaftskonzept im Sprachumfang definiert haben, wäre in diesem Fall eine Compiler-Erweiterung erforderlich. Ein Weg, dies zu vermeiden, besteht darin, Kommunikationsvorgänge (dazu gehören auch die ursprünglichen Eingaben und die Ausgaben für die Gesamtergebnisse) über eigens definierte Prozeduren in fremden Modulen zu formulieren und im eigentlichen Transformationsmodul an den Kommunikationsstellen lediglich die definierten Prozeduren aufzurufen. Wenn jede Eingabe einer Teilaufgabe über eine Eingabeprozedur angefordert wird und jede Ausgabe über eine Ausgabeprozedur abgesetzt wird, ist auch eine Modellierung von Datenströmen adäquat möglich. Im folgenden sollen derartige Ein- und Ausgabeprozeduren als Kommunikationsprozeduren bezeichnet werden.

Damit ergibt sich als erster Umformungsschritt die Überführung einer Teilaufgabe in eine E/A-orientierte Teilaufgabe. Dabei gelten in Anlehnung an den in [Klei82] definierten Begriff der Elementaraufgabe die folgenden Definitionen:

Def. 2.6: Eine Teilaufgabe ist definiert durch ein r -tupel von Eingabedaten, ein q -tupel von Ausgabedaten und eine Berechnungsvorschrift, die für jedes r -tupel von Eingabedaten ein q -tupel von Ausgabedaten erzeugt.

Im folgenden werden die Begriffe "Eingabedaten" und "Eingaben" bzw. "Ausgabedaten" und "Ausgaben" synonym verwendet.

Def. 2.7: Unter einer E/A-orientierten Teilaufgabe versteht man eine Teilaufgabe, bei denen benötigte Eingaben über Kommunikationsprozeduren angefordert werden und erzielte Ausgaben über Kommunikationsprozeduren abgesetzt werden.

Zur Illustration der Transformation sei folgendes Programmbeispiel betrachtet.

Eine Menge von Zahlen, welche in einem Feld abgespeichert sind, soll sortiert werden. Eine Lösung in Form eines "klassischen" Sortiermoduls zeigt Abb. 2.5. Die direkte Umsetzung dieses

```
MODULE KSORT
/*      -----      "KLASSISCH" SORTIEREN      -----      */
GLOBAL SUBR KSORT = (INT DIMENSION, REF [] INT ZAHLFELD)
BEGIN
  INT MININD;
  FOR I TO DIMENSION-1
  DO MININD := I;
    FOR J FROM I TO DIMENSION
    DO IF ZAHLFELD[MININD] > ZAHLFELD[J]
      THEN MININD := J
      FI
    OD;
    IF I /= MININD
    THEN ZAHLFELD[I] :: ZAHLFELD[MININD]
    FI
  OD
END
MODEND
```

Abb. 2.5: Ausgangsbeispiel eines Moduls für die Aufgabe "Sortierung einer Zahlenfolge"

Sortiermoduls in einen E/A-orientierten Sortiermodul führt zu der in Abb. 2.6 dargestellten Lösung.

Die Routinen EINGABE und AUSGABE sind entsprechend den Anforderungen der Umgebung formulierbar (z. B. als Einlesen von einer Datei bzw. Ausschreiben auf eine Datei oder auf einen im Hauptspeicher realisierten Datenpuffer).

```

MODULE EASORT

/*      -----      AUSSENBEZUEGE FUER E/A      -----      */

VALGLOB SUBR (REF INT) EINGABE;
VALGLOB SUBR (INT) AUSGABE;

/*      -----      E/A-ORIENTIERT SORTIEREN      -----      */

GLOBAL SUBR EASORT = (INT DIMENSION)

BEGIN
  [1:DIMENSION] INT ZAHLFELD;
  INT MININD;

  FOR I TO DIMENSION
  DO EINGABE(ZAHLFELD[I])
  OD;

  FOR I TO DIMENSION-1
  DO MININD := I;
    FOR J FROM I TO DIMENSION
    DO IF ZAHLFELD[MININD] > ZAHLFELD[J]
      THEN MININD := J
    FI
  OD;
  IF I /= MININD
  THEN ZAHLFELD[I] :: ZAHLFELD[MININD]
  FI
  OD;

  FOR I TO DIMENSION
  DO AUSGABE(ZAHLFELD[I])
  OD
END

MODEND

```

Abb. 2.6: Beispielmol für die Aufgabe "Sortierung einer Zahlenfolge E/A-orientiert"

Im allgemeinen fordert eine Teilaufgabe von mehr als einer Stelle Eingaben an bzw. setzt an mehr als eine Stelle Ergebnisse ab. Man könnte nun für jeden dieser verschiedenen E/A-Vorgänge eine eigene Kommunikationsprozedur formulieren und implementieren. Dies ist jedoch nicht beabsichtigt. Stattdessen wird versucht, "formatgleiche" Ein- bzw. Ausgaben zusammenfassen.

In Programmiersprachen werden Eingaben von verschiedenen Dateien bzw. Ausgaben auf verschiedene Dateien über Kanalnummern unterschieden. Verschiedenen Dateien sind verschiedene Kanalnummern zugeordnet. Kanalnummern zur Ein- bzw. Ausgabe auf verschiedene Datenstrukturen sind direkt in obigem Schema nachbildbar.

Eine zweite noch nicht nachgebildete Eigenschaft von E/A-Anweisungen in Programmiersprachen ist die der Behandlung von bei diesen Routinen auftretenden besonderen Ereignissen. Eingaben werden im allgemeinen dadurch beendet, daß nach Einlesen des letzten Zeichens bei nachfolgenden Leseanweisungen eine Quittung EOF zurückgeliefert wird, bzw. eine Fehleranzeige, wenn beim Lesen ein Fehler auftrat. Ausgaben auf Dateien werden mit einem Zeichen quittiert, welches anzeigt, ob die Ausgabe korrekt erfolgte oder ob bei der Ausgabe ein Fehler auftrat (z. B. "Ausgabe in nicht zulässigem Format" oder "Ausgabedatei zu klein").

Diese Eigenschaften werden durch E/A-orientierte Teilaufgaben mit Kontrolle nachgebildet, die wie folgt definiert werden.

Def. 2.8: Eine E/A-orientierte Teilaufgabe mit Kontrolle ist eine E/A-orientierte Teilaufgabe, bei der die Ein- und Ausgabeparameter aus Kanalnummern und Quittungszeichen bestehen. Kanalnummern dienen zur Unterscheidung verschiedener Ein- bzw. Ausgabebereiche, Quittungszeichen dienen dazu, die Art der Ausführung (korrekt oder fehlerhaft) anzuzeigen.

Dem Aufrufer einer solchen Teilaufgabe wird dabei überlassen, wie er den weiteren Programmablauf in Abhängigkeit der erhaltenen Quittung steuert.

In der Regel werden Prozedurquittungen in Form von ganzzahligen Größen an das aufrufende Programm zurückgemeldet. Abb. 2.7 zeigt die Formulierung des Sortiermoduls als E/A-orientierte Teilaufgabe mit Kontrolle.

Dabei seien folgende Kontrollrückgaben vereinbart:

- 0...korrekter Ablauf
- 1...Fehler bei Eingabe
- 2...Fehler bei Ausgabe

Die Routine EINGABE möge bei korrekter Ausführung die Quittung 0¹¹⁾, bei Datenende die Quittung EOF (als einen systemspezifischen Wert), beide Routinen bei fehlerhafter Ausführung einen von diesen Größen verschiedenen Wert zurückmelden, der im dargestellten Fall nicht weiter aufgeschlüsselt wird.

Eine direkte Formulierung der Behandlung besonderer Ereignisse innerhalb eines Programmlaufes, wie sie etwa innerhalb der Programmiersprachen CLU [LiSn79] oder PL/I [MacL77] realisiert ist, wäre sicher wünschenswert, wird jedoch aus folgenden Gründen nicht nachgebildet:

1. In allgemein verwendeten Programmiersprachen wäre zu diesem Zweck eine Compiler-Erweiterung erforderlich. Lösungen, wie sie in C [Lee83] durch Präprozessoren realisiert werden, können nicht auf beliebige Programmiersprachen übertragen werden. Dagegen ist der Quittungsmechanismus in beliebigen Sprachen nachbildbar.
2. Eine solche Behandlung läßt sich nicht problemlos von sequentiellen in Parallelprogramme (gemäß Def. 2.5) übertragen. In Parallelprogrammen wird sicher eine prozessorspezifische Lösung dieser Ereignisse vorzunehmen sein. Eine auf der "sequentiellen" Ebene formulierte Lösung würde eine für Parallelprogramme unnötige Aufgabe flexibler Behandlungsmethoden bedeuten.

¹¹⁾ Bei korrekter Ausführung einer E/A-Routine wird neben der Quittung im allgemeinen auch zurückgemeldet, wieviele Zeichen (Bytes) übertragen wurden (vgl. Kap. 6).

```

MODULE EAKSORT

/*      -----      AUSSENBEZUEGE FUER E/A      -----      */

VALGLOB SUBR (INT, REF INT) INT EINGABE;
VALGLOB SUBR (INT, INT) INT AUSGABE;

/*      -----      E/A-ORIENTIERT SORTIEREN      -----      */
      MIT KONTROLLE

GLOBAL SUBR EAKSORT = (INT INKANAL, OUTKANAL) INT

BEGIN
  INT DIMENSION $= "Zahl sortierbarer Elemente";
  INT EOF $= "Systemdarstellung für Datenende";
  [1:DIMENSION] INT ZAHLFELD;
  INT LAUFIND, MININD, QUITTUNG;

  LAUFIND := QUITTUNG := 0;
  WHILE QUITTUNG /= EOF
  DO IF LAUFIND >= DIMENSION THEN RETURN(-1) FI;
    LAUFIND := LAUFIND + 1;
    QUITTUNG := EINGABE(INKANAL, ZAHLFELD[LAUFIND]);
    IF QUITTUNG /= 0 AND QUITTUNG /= .EOF THEN RETURN(-1) FI
  OD;

  FOR I TO LAUFIND-1
  DO MININD := I;
    FOR J FROM I TO LAUFIND
    DO IF ZAHLFELD[MININD] > ZAHLFELD[J]
      THEN MININD := J
    FI
  OD;
  IF I /= MININD
  THEN ZAHLFELD[I] := ZAHLFELD[MININD]
  FI
  OD;

  FOR I TO LAUFIND
  DO QUITTUNG := AUSGABE(OUTKANAL, ZAHLFELD[I]);
    IF QUITTUNG /= 0 THEN RETURN(-2) FI
  OD;
  RETURN(0)
END

MODEND

```

Abb. 2.7: Beispiel eines Moduls für die Aufgabe "Sortierung einer Zahlenfolge E/A-orientiert mit Kontrolle"

Auf Fragen der Behandlung besonderer Ereignisse in Parallelprogrammen wird in Kap. 5.3.5 eingegangen.

Eine Formulierung von Modulen, die Ergebnisse über selbst definierte Kommunikationsprozeduren austauschen, bietet auch im Bereich der sequentiellen Programmierung verschiedene Vorteile.

Auf diese Weise können Module nicht nur unabhängig voneinander implementiert werden, sondern auch auf einfache Art getestet werden. Zu diesem Zweck sind lediglich die Kommunikationsprozeduren entsprechend zu formulieren (z. B. so, daß Eingaben im Dialog vom Benutzer angefordert werden bzw. Ausgaben in für den Benutzer interpretierbarer Form abgesetzt werden).

Bei einer Realisierung der Kommunikationsprozeduren als Einlesen von Dateien bzw. Ausgabe auf Dateien können in Fehlerfällen aus der Analyse der bis zu diesem Zeitpunkt aufgetretenen Ausgaben häufig schon recht weitgehende Schlüsse auf Fehlerursachen gezogen werden.

Wesentlichster Vorteil gerade im Hinblick auf Parallelprogrammierung ist jedoch, daß auf dieser Ebene auch die Formulierung von verschiedenen Formen von Prozessorkommunikation möglich ist (vgl. Kap. 5.2). Einige Systeme auf Multiprozessoren und Konzepte für Parallelismus haben E/A-Primitiva direkt als Sprachmittel für die Prozessorkommunikation implementiert bzw. vorgeschlagen, z. B. [Han78], [Hoa78], [KaMQ77], unter anderem auch das UNIX-System [RiTh74]. In Kap. 5.1.2 wird eine Möglichkeit zur Implementierung von Parallelprogrammen auf der Basis des UNIX-Systems vorgestellt.

Abschließend sei noch auf zwei Prinzipien paralleler Programmierung hingewiesen, die mit Kommunikationsprozeduren als Mittel zum Austausch von Informationen zwischen verschiedenen Modulen realisiert werden können. Das eine kann als Prinzip "Eingaben so spät wie möglich", das andere als Prinzip "Ausgaben so früh wie möglich" charakterisiert werden.

"Eingaben so spät wie möglich" gestaltet ein Programm so, daß so viele Aktionen wie möglich mit einer erhaltenen Eingabe ausgeführt werden, bevor eine neue Eingabe angefordert wird. "Ausgaben so früh wie möglich" gestaltet ein Programm so, daß Ergebnisse zum frühest möglichen Zeitpunkt berechnet und ausgegeben werden.

Die Anwendung dieser Prinzipien findet vor allem in Parallelprogrammen Anwendung, die als Datenflußprogramme konstruiert sind (vgl. etwa Kap. 3 und 4), d. h. in Programmen, in denen der Programmablauf auf verschiedenen Prozessoren in Abhängigkeit der Verfügbarkeit von Daten gesteuert wird.

Eine Eingabeforderung bezgl. eines nicht verfügbaren Datums führt in der Regel auf einen Wartezustand ("Warten, bis angefordertes Datum verfügbar"). "Eingaben so spät wie möglich" versucht, einen solchen potentiellen Wartezustand zu vermeiden bzw. eine potentiell mögliche Wartezeit zu minimieren, indem neue Daten so spät wie möglich, d. h. erst unmittelbar dann angefordert werden, wenn sie wirklich benötigt werden.

Das Absetzen eines erzielten Ergebnisses führt möglicherweise dazu, daß ein bisher darauf wartender Prozeß seine Arbeit fortsetzen kann. "Ausgaben so früh wie möglich" versucht, eine potentielle Wartezeit eines anderen (auf dieses Ergebnis) wartenden Prozesses zu minimieren, indem erzielte Ergebnisse so früh wie möglich, d. h. unmittelbar nach dem Zeitpunkt ihres Entstehens, ausgegeben werden.

Parallele Verarbeitungsstrategien, die zum Ziel haben, die Summe der Wartezeiten und damit auch die Gesamtausführungszeit eines Parallelprogramms zu minimieren (vgl. Kap. 3.2.1), versuchen, möglichst beide Prinzipien zu realisieren.

Ohne eine formale Anleitung zur Realisierung dieser Prinzipien in Programm-Modulen zu geben, seien hier die beiden Prinzipien anhand des Beispiels "Sortierung" illustriert. Abb. 2.8 gibt eine Programmformulierung nach dem Prinzip "Eingaben so spät wie möglich". Im Fall der Sortierung wird bei der Eingabe die

erhaltene Zahl sofort innerhalb des Feldes an der richtigen Stelle eingeordnet und eine neue Eingabe erst dann angefordert, wenn die aktuelle Eingabe nach diesem Verfahren vollständig "verarbeitet" ist.

Abb. 2.9 gibt eine Programmformulierung nach dem Prinzip "Ausgaben so früh wie möglich". Eine Ausgabe erfolgt in dem Moment, in dem ein neues Minimum im noch zu sortierenden Zahlenbereich gefunden ist.

Am Anwendungsbeispiel "Netzwerkanalyse" in Kap. 4.2 wird die Bedeutung der beiden Prinzipien in Parallelprogrammen verdeutlicht.

MODULE EASORTESM

```

/*      -----      AUSSENBEZUEGE FUER E/A      -----      */
VALGLOB SUBR (INT, REF INT) INT EINGABE;
VALGLOB SUBR (INT, INT) INT AUSGABE;

/*      -----      E/A-ORIENTIERT SORTIEREN,      -----      */
      EINGABEN SO SPÄT WIE MÖGLICH
GLOBAL SUBR ESORTESM = (INT INKANAL, OUTKANAL) INT

BEGIN
  INT DIMENSION $= "Zahl sortierbarer Elemente";
  INT EOF $= "Systemdarstellung für Datenende";
  [1:DIMENSION] INT ZAHLFELD;
  INT LAUFIND, MININD, QUITTUNG, PLATZ, WERT;

  QUITTUNG := LAUFIND := 0;
  WHILE QUITTUNG /= EOF
  DO IF LAUFIND >= DIMENSION THEN RETURN(-1) FI;
    QUITTUNG := EINGABE(INKANAL, WERT);
    IF QUITTUNG /= 0 AND QUITTUNG /= EOF THEN RETURN(-1) FI;

    /*EINORDNEN*/
    PLATZ := LAUFIND + 1;
    FOR I TO LAUFIND
    DO IF WERT < ZAHLFELD[I]
      THEN PLATZ := I; BREAK
      FI
    OD;
    LAUFIND := LAUFIND + 1;
    IF PLATZ < LAUFIND
    THEN FOR I FROM LAUFIND BY -1 TO PLATZ+1
      DO ZAHLFELD[I] := ZAHLFELD[I-1]
      OD
    FI;
    ZAHLFELD[PLATZ] := WERT
    /*EINORDNEN BEENDET*/
  OD;

  FOR I TO LAUFIND
  DO QUITTUNG := AUSGABE(OUTKANAL, ZAHLFELD[I]);
    IF QUITTUNG /= 0 THEN RETURN(-2) FI
  OD;
  RETURN(0)
END
MODEND

```

Abb. 2.8: Beispiel eines Moduls für die Sortierung nach dem Prinzip "Eingaben so spät wie möglich"

MODULE EASORTAFM

```

/*      -----      AUSSENBEZUEGE FUER E/A      -----      */
VALGLOB SUBR (INT, REF INT) INT EINGABE;
VALGLOB SUBR (INT, INT) INT AUSGABE;

/*      -----      E/A-ORIENTIERT SORTIEREN,
      AUSGABEN SO FRÜH WIE MÖGLICH      -----      */

GLOBAL SUBR EASORTAFM = (INT INKANAL, OUTKANAL) INT
BEGIN
  INT DIMENSION $= "Zahl sortierbarer Elemente";
  [1:DIMENSION] INT ZAHLFELD;
  INT LAUFIND, MININD, QUITTUNG;

  QUITTUNG := LAUFIND := 0;
  WHILE QUITTUNG /= EOF
  DO IF LAUFIND >= DIMENSION THEN RETURN(-1) FI;
    LAUFIND := LAUFIND + 1;
    QUITTUNG := EINGABE(INKANAL, ZAHLFELD[LAUFIND]);
    IF QUITTUNG /= 0 AND QUITTUNG /= EOF THEN RETURN(-1) FI;
  OD;

  FOR I TO LAUFIND-1
  DO MININD := I;
    FOR J FROM I TO LAUFIND
    DO IF ZAHLFELD[MININD] > ZAHLFELD[J]
      THEN MININD := J
      FI
    OD;
    QUITTUNG := AUSGABE(OUTKANAL, ZAHLFELD[MININD]);
    IF QUITTUNG /= 0 THEN RETURN(-2) FI;
    IF I /= MININD
    THEN ZAHLFELD[I] := ZAHLFELD[MININD]
    FI
  OD;

  QUITTUNG := AUSGABE(OUTKANAL, ZAHLFELD[LAUFIND]);
  IF QUITTUNG /= 0 THEN RETURN(-2) FI;
  RETURN(0)

END
MODEND

```

Abb. 2.9: Beispiel eines Moduls für die Sortierung nach dem Prinzip "Ausgaben so früh wie möglich"

2.3 Zur Teilung der Datenmenge

Eine Zuordnung ganzer Teilaufgaben, die nach den Richtlinien des Kap. 2.2 gebildet wurden, auf Prozessoren eines Multiprozessorsystems ist nur möglich, wenn die Zahl der Teilaufgaben nicht die Zahl der verfügbaren Prozessoren übersteigt und die Abhängigkeit bestimmter Teilaufgaben voneinander auf die Struktur des Multiprozessorsystems abgebildet werden kann (vgl. hierzu Kap. 3.1). Im allgemeinen werden in Multiprozessorsystemen mit eingeschränkten Ressourcen für ein gegebenes Programmsystem jedoch mehr Teilaufgaben als Prozessoren existieren. Um dann trotzdem die Verfahren des Kapitels 2.2 anwenden zu können, ist eine Zusammenfassung mehrerer Teilaufgaben auf einem Prozessor erforderlich.

Wenn die modulare Struktur der Teilaufgaben des Kapitels 2.2 dabei nicht zerschlagen werden soll, ergibt sich das Problem der Zwischenspeicherung von Ergebnissen abgelaufener Teilaufgaben.

Es läßt sich zwar denken, daß eine Teilaufgabe die von ihr erzielten Ergebnisse auf eine Datei schreibt, die von der diese Ergebnisse weiterverarbeitenden Teilaufgabe gelesen werden. Dieses Verfahren ist jedoch nur anwendbar, wenn der Prozessor, dem die entsprechenden Teilaufgaben zugeordnet sind, von Dateien lesen und auf Dateien schreiben kann (d. h. zumindest mit der entsprechenden Peripherie verbunden ist). Selbst wenn dieses Verfahren möglich ist, müßte es aus Effizienzgründen abgelehnt werden, da häufiges Ein- und Ausgeben die Programmlaufzeiten signifikant verlängert. Kap. 4.1 gibt ein Beispiel.

Zwischenergebnisse müssen demnach im Hauptspeicher gehalten werden. Dazu sind Datenstrukturen zu deklarieren, die in Form abstrakter Datentypen angelegt sein können. Implementierungsmöglichkeiten solcher abstrakter Datentypen in Abhängigkeit der Speichertechnologie (als adressierbarer bzw. assoziativer Speicher) finden sich in Kap. 5.2. Beschreibungen zum Konzept und zur Programmierung finden sich in [GuHo78] und [LiZi74]. Programmiersprachen, die das Konzept in seiner Definition als

Erweiterung des Typkonzepts in Programmiersprachen enthalten, sind etwa CLU [LiSn77] oder ALPHARD [WLS76]. Die Nachbildung der abstrakten Datentypen auf der Ebene der Maschinensprache ist die Grundlage der im STARLET-Rechner realisierten Datentypen-Architektur [GiGu82].

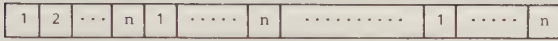
Wenn Ergebnisse von Teilaufgaben auf dem Hauptspeicher abgelegt werden sollen, ergibt sich das Problem, daß der verfügbare Speicherraum nicht ausreicht, um die entstehende Datenmenge aufzunehmen. Es ist auch möglich, daß die entstehende Datenmenge zwar im Hauptspeicher Platz findet, aber keine Möglichkeit besteht, einen Bereich der notwendigen Größe zu deklarieren.¹²⁾

In beiden Fällen ist die Aufgabe jedoch in der modularen Form des Kap. 2.2 lösbar, wenn die Menge der Daten, die eine Teilaufgabe zu verarbeiten hat, in geeignet kleine Blöcke aufgeteilt werden kann.

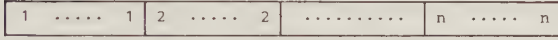
Einfach erkennbare Aufteilungen der Datenmenge ergeben sich, wenn dem zu lösenden Problem in der Realität ein Netzwerk aus miteinander verbundenen Bausteinen zugrundeliegt (wie z. B. dem Anwendungsbeispiel "Netzwerkanalyse" des Kap. 4.2). Dann kann jeder Baustein mit seinen zu verarbeitenden Daten einem Prozessor des Multiprozessorsystems zugeordnet werden. Wenn weniger Prozessoren als Einzelbausteine vorhanden sind, werden Gruppen von Bausteinen einzelnen Prozessoren zugeordnet.

Eine andere Ebene für ein Vorgehen zur Aufteilung von zu verarbeitenden Daten auf Prozessoren eines Multiprozessorsystems ergibt sich, wenn der zugrundeliegende Algorithmus als eine Menge von Transformationen auf Vektoren und Matrizen formuliert ist. Dann ist in der Regel eine Zuordnung von Einzelelementen bzw. Gruppen von Einzelelementen der Datenstrukturen auf einzelne Prozessoren möglich. Abb. 2.10 zeigt zwei Formen einer

12) Im für die praktischen Implementierungen zugrundeliegenden EGPA-System mit der Programmiersprache SL3 ist es z. B. nicht möglich, Datenstrukturen zu deklarieren, die mehr als 64 K Speicherplatz einnehmen.



Elementweise Zerlegung



Mengenweise Zerlegung

Abb. 2.10: Formen einer Zuordnung von Vektorelementen zu Prozessoren eines Multiprozessorsystems



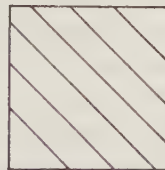
Quadrantenzerlegung



Streifenzerlegung
(waagerecht oder senkrecht)



Sektorzerlegung



Zerlegung in Diagonalstreifen

Abb. 2.11: Formen einer Zuordnung von Matrixelementen zu Prozessoren eines Multiprozessorsystems

gruppenweise Aufteilung von Vektorelementen, Abb. 2.11 einige Formen einer gruppenweise Aufteilung von Matrixelementen.¹³⁾

Es läßt sich nicht generell sagen, welche Aufteilungsform für ein gegebenes Problem bzw. eine Klasse von Problemen die günstigste ist. Kriterien einer guten Aufteilung sind, daß möglichst wenige Zugriffe auf Daten fremder Prozessoren erforderlich werden, daß möglichst wenige Transportoperationen von Daten von einem Prozessor zu einem anderen Prozessor notwendig sind und daß möglichst wenige Daten mehrfach gehalten werden müssen.

Häufig sind mehrere Formen der Datenaufteilung¹⁴⁾ möglich. In Kap. 4.2 wird am Beispiel der Netzwerkanalyse die Güte einiger Möglichkeiten der Datenaufteilung bewertet. Praktisch alle parallelen Algorithmen der numerischen Mathematik gehen von einer Aufteilung der zu verarbeitenden Datenmenge auf Prozessoren eines Multiprozessorsystems aus.

Im Fall ordnungserhaltender Programmsysteme ist ein Arbeiten auf Strukturen von Vektoren oder Matrizen von ganzen oder reellwertigen Zahlen nicht der Regelfall. Hier legt die Form der Eingabe von Daten und ihre anschließende Speicherung im Rechner eine Art Vektorstruktur fest. Dieser Vektor besteht jedoch nicht mehr aus Zahlenelementen, sondern aus Objekten "höheren" Typs. Im Fall der Compiler [Mes82] oder der Textformatierung (vgl. Kap. 4.1) sind dies Einheiten von abgeschlossenen Stringzeilen, bei der Verarbeitung von Meßwerten [KaKo83] in sich abgeschlossene Zahlengruppen, bei Interpretern in sich abgeschlossene Symbolketten oder Fragmente usw. In all diesen Fällen ist die Datenaufteilung allein aufgrund der Struktur der einlaufenden Datenmenge nicht a priori möglich, sondern wird

- 13) Die gruppenweise Aufteilung von Matrixelementen parallel zur Hauptdiagonalen ist eine, die der Wellenfront-Methode zur Parallelisierung von DO-Loops des Kap. 2.1.2 angepaßt ist.
- 14) Neben den bereits genannten Gesichtspunkten können auch Fragen möglicher Formen von Speicherzugriffen (adreßbezogen, assoziativ) die Art der Datenaufteilung beeinflussen.

durch die Arbeitsweise der die Daten transformierenden Programme bestimmt.

Die Überlegung, welche Mindestmenge an Eingabedaten notwendig ist, um ein Programm testen zu können, kann als Richtlinie zur Aufteilung der zu verarbeitenden Datenmenge dienen. Diese ist gleichbedeutend mit der minimalen Menge an Eingabedaten, die eine als "richtig" oder "falsch" im Sinn der Aufgabe zu klassifizierende Menge an Ausgabedaten erzeugt. Für einen Compiler/Code-generator sind dies beispielsweise vollständige Anweisungen im Quellprogramm bzw. Befehle in der Zwischensprache.

Eine Aufteilung eines Datenbestandes in Blöcke, die minimale Arbeitsmengen enthalten, ist korrekt, wenn die vollständig transformierten Daten der verschiedenen Blöcke am Ende zum korrekten Gesamtergebnis zusammengesetzt werden können. Derartige Folgen von Datenblöcken sollen informationskonsistent genannt werden.

Def. 2.9: Eine Folge von Datenblöcken $D_1, D_2, \dots, D_m$ aus Eingabedaten heißt informationskonsistent bezgl. einer ordnungserhaltenden Transformation f , wenn für die Konkatenation der einzelnen Datenblöcke (bezeichnet durch " $\circ$ ") die Homomorphiebedingung

$$f(D_1 \circ D_2 \circ \dots \circ D_m) = f(D_1) \circ f(D_2) \circ \dots \circ f(D_m)$$

erfüllt ist.

Die Informationskonsistenz einer Folge von Datenblöcken bezgl. einer ordnungserhaltenden Transformation besagt also, daß für die Konkatenation der Datenblöcke das Ergebnis der Programmtransformation unabhängig davon ist, ob die Konkatenation der (ursprünglichen) Datenblöcke transformiert wird oder die (ursprünglichen) Datenblöcke einzeln transformiert und erst dann konkateniert werden.

Daß informationskonsistente Folgen von Datenblöcken konstruiert werden können, hängt übrigens wesentlich davon ab, daß die im Zusammenhang damit betrachtete Transformation ordnungserhaltend ist.

Im Beispiel der Textformatierung (vgl. Kap. 4.1) gilt Informationskonsistenz, wenn von Absätzen als Grundlage von Datenblöcken ausgegangen wird, nicht jedoch, wenn Stringzeilen oder einzelne Wörter die Grundlage von Datenblöcken bilden. (In letzterem Fall ergeben sich für den Zeilenumbruch mit Randausgleich bei Anwendung der beiden in Def. 2.9 angesprochenen Verfahren unterschiedliche Ergebnisse.)

Wie groß die minimale Arbeitsmenge ist und wie unabhängig voneinander Datenblöcke verarbeitet werden können, die aus derartigen Portionen gebildet sind, entscheidet letztlich über die Möglichkeiten, Parallelisierungsstrategien auf der Grundlage von Datenaufteilung zu entwickeln bzw. die Nicht-Parallelisierbarkeit eines Programms auf dieser Basis festzustellen.

2.4 Die Entflechtung von Teilaufgaben

Wenn ein Programm in Teilaufgaben partitioniert ist, die in der Lage sind, auf informationskonsistenten Folgen von Datenblöcken zu arbeiten, ist eine Darstellung der Abhängigkeiten zwischen den Teilaufgaben durch den in Abb. 2.12 skizzierten Graphen beschreibbar. Dabei kann die mit t_1 bezeichnete Teilaufgabe als "Eingabe des ursprünglichen Datenstroms", die mit t_n bezeichnete Teilaufgabe als "Ausgabe des schließlich erzielten Ergebnisses" interpretiert werden.

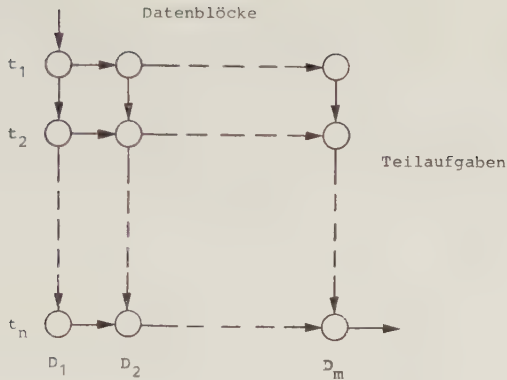


Abb. 2.12: Graph der Abhängigkeiten von Teilaufgaben eines ordnungserhaltenden Programmsystems

Für die Abarbeitung innerhalb eines sequentiellen Programms bieten sich die beiden folgenden Alternativen an:

1. "Horizontale" Verarbeitungsrichtung, d. h. für die gesamte Datenmenge wird zunächst t_1 durchlaufen und nach Ausführung von t_i mit t_{i+1} (für $1 \leq i < n$) fortgefahren. Dies entspricht der in Kap. 2.2 beschriebenen Arbeitsweise. Diese hat den Vorzug, daß dabei Überlegungen zur Aufteilung der Datenmenge nicht erforderlich sind und ein funktionsweiser Programmtest über den gesamten Datenbereich möglich ist. Nachteil ist, daß Zwischenergebnisse wegen des Umfangs der Gesamtdatenmenge sicher nicht allgemein im Hauptspeicher gehalten werden können, sondern beispielsweise auf Dateien geschrieben bzw. von Dateien gelesen werden müssen.
2. "Vertikale" Verarbeitungsrichtung, d. h. es wird zunächst für den ersten Datenblock die gesamte Folge von Teilaufgaben durchlaufen, und nach der vollständigen Transformation des i -ten Datenblocks mit der vollständigen Transformation des

(i+1)-ten Datenblocks fortgeföhren. Diese Arbeitsweise ermöglicht im allgemeinen, die Zwischenergebnisse im Hauptspeicher zu halten und damit "zeitvergeudende" E/A-Aufrufe einzusparen. Nachteil ist jedoch, daß zuvor die Datenmenge in Blöcke aufgeteilt werden muß, die bei Zusammenfügen der Einzelergebnisse das "richtige" Gesamtergebnis liefern muß.

Die Abbildungen 2.13 und 2.14 symbolisieren die beiden Verarbeitungsrichtungen.¹⁵⁾

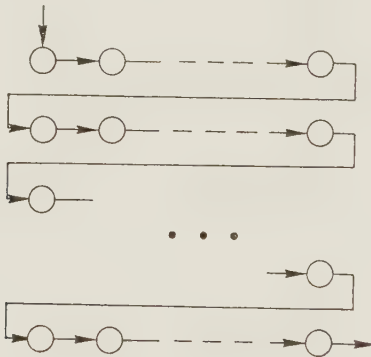


Abb. 2.13: Horizontale Verarbeitung des Teilaufgabengraphen der Abb. 2.12

Zwischen den beiden skizzierten Verarbeitungsalternativen liegen eine Menge anderer zulässiger Abarbeitungsreihenfolgen, die bei

- 15) Bei einer Lohn- und Gehaltsrechnung für Mitarbeiter eines Unternehmens besteht die horizontale Verarbeitungsrichtung darin, zunächst für alle Mitarbeiter zusammen das Bruttogehalt zu berechnen, dann für alle Mitarbeiter die Lohnsteuer, dann für alle Mitarbeiter die Sozialabgaben usw. Im gleichen Fall bedeutet die vertikale Verarbeitungsrichtung, für jeden Mitarbeiter einzeln sämtliche Lohnkomponenten (bzw. das Nettogehalt) zu berechnen, bevor Lohn und Gehalt für den nächsten Mitarbeiter berechnet werden.

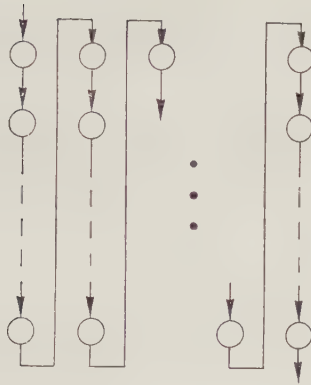


Abb. 2.14: Vertikale Verarbeitung des Teilaufgabengraphen der Abb. 2.12

der Beschreibung paralleler Verarbeitungsstrategien eine Rolle spielen (vgl. Kap. 3). Diese fassen Bündel von Funktionen bzw. Bündel von Datenblöcken zusammen. Eine maximal parallele Verarbeitungsform, etwa mit den Methoden des Kap. 2.1.1 erzeugt, würde auf die in Kap. 2.1.2 beschriebene Wellenfront-Methode¹⁶⁾ führen. Parallele Verarbeitungsstrategien des Kap. 3 stützen sich jedoch nicht auf die Wellenfront-Methode ab, da sie einen sehr hohen Aufwand an Kommunikation und Fremdspeicherzugriffen zwischen verschiedenen Prozessoren bedeutet.

Parallele Verarbeitungsstrategien auf der Grundlage von Datenaufteilung in Kap. 3 nehmen als Ausgangspunkt eine Beschreibung der zulässigen Abarbeitungsreihenfolgen in Form eines Graphen, in dem möglichst viele Abhängigkeiten zwischen verschiedenen Teilaufgaben abgebaut sind, um für die spätere Zuord-

16) In diesem Fall würde sich ein makro-systolischer paralleler Algorithmus von der Art ergeben, wie er in [HBFHV84] für die Matrixmultiplikation dargestellt ist.

nung einzelner Teilaufgaben zu Prozessoren eines Multiprozessor-systems Kommunikationen und Fremdspeicherzugriffe zu minimieren. Die Entflechtung von Teilaufgaben hat zum Ziel, solche Abhängigkeiten abzubauen.

Def. 2.10: Unter Entflechtung zweier voneinander bezgl. der Abarbeitungsreihenfolge abhängiger Teilaufgaben versteht man eine Folge von Maßnahmen, nach deren Ausführung diese Abhängigkeit beseitigt ist.

Nach Beseitigung von Abhängigkeiten bezgl. der Abarbeitungsreihenfolge von Teilaufgaben ist dann eine Ausführung der betreffenden Teilaufgaben in beliebiger Reihenfolge und damit eine parallele Ausführung dieser Teilaufgaben möglich.

In seiner Zielrichtung entspricht die Entflechtung der in [Wal82] angestrebten Form der Parallelisierung, nämlich die Menge der sequentiell auszuführenden Programmteile zu minimieren und die Menge der parallel ausführbaren Programmteile zu maximieren.

Beim Abbau von Abhängigkeiten bezgl. der Teilaufgabenordnung des Graphen der Abb. 2.12 ergeben sich einige Vorgaben, die dies verhindern:

1. Abhängigkeiten zwischen aufeinanderfolgenden Eingaben von Datenblöcken und zwischen aufeinanderfolgenden Ausgaben von Datenblöcken müssen erhalten bleiben (vgl. Def. des ordnungserhaltenden Programmsystems in Kap. 1.1).
2. Abhängigkeiten zwischen aufeinanderfolgenden Transformationen, die den gleichen Datenblock betreffen, müssen erhalten bleiben.

Der noch "erlaubte" Abbau von Abhängigkeiten zwischen Teilaufgaben führt formal den Graphen der Abb. 2.12 in den in Abb. 2.15 über.

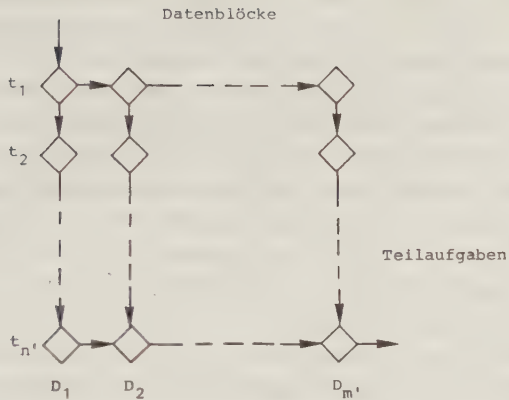


Abb. 2.15: Entflochtener Graph der Abhängigkeiten von Teilaufgaben eines ordnungserhaltenden Programmsystems

In vielen Fällen sind die durch Datenteilung entstandenen Teilaufgaben im Sinne des Graphen der Abb. 2.15 bereits mit der Partitionierung entflochten. Ist dies nicht der Fall, kann evtl. eine Entflechtung durchgeführt werden, indem Teilaufgaben modifiziert werden, neue Teilaufgaben hinzukommen und/oder die Datenmenge nach "neuen" Kriterien partitioniert wird.

Derartige Maßnahmen müssen im Zusammenhang damit gesehen werden, daß sich Abhängigkeiten zwischen Teilaufgaben in Wirklichkeit nicht beseitigen, sondern bestenfalls so in die Teilaufgaben hinein verlagern lassen, daß zuvor noch zwischen Teilaufgaben bestehende Abhängigkeiten danach nach außen nicht mehr sichtbar sind.

Dies bedeutet bezogen auf die Abb. 2.12 und 2.15, daß eine Entflechtung nicht einfach dadurch stattfindet, daß die waagrechten Pfeile in den zwischen "Eingabe" und "Ausgabe" liegenden Teilaufgaben gestrichen werden, sondern daß dazu vom Programmierer durchzuführende Änderungen am Algorithmus erforderlich sein können, durch die Zahl und Struktur von Teilaufgaben bzw. zu transformierenden Datenblöcken verändert werden. Unter Umständen (vgl. einführende Programmbeispiele des Kap. 1.1) kann sich die Forderung "ordnungserhaltend" als zu stark für die Anwendbarkeit von Entflechtungsmaßnahmen und damit letztlich für die darauf aufbauende parallele Verarbeitung von Teilaufgaben erweisen.

Die Korrektheit einer durchgeführten Entflechtung läßt sich auf einfache Weise nachprüfen. Man führt die verschiedenen Teilaufgaben in der oben als "vertikale Verarbeitungsweise" bezeichneten Art aus, (d. h. für einen Datenblock alle Funktionen, bevor der nächste Datenblock bearbeitet wird), wobei zu jedem Start einer Teilaufgabe (mit Ausnahme der Teilaufgaben "Eingabe" und "Ausgabe") die Initialisierungen für die entsprechende Funktion mit ausgeführt werden. Damit ist sichergestellt, daß bei der Ausführung einer Teilaufgabe Aktionen aus der Vergangenheit des Programms keinen Einfluß mehr haben. Die Entflechtung ist dann korrekt, wenn mit dieser Arbeitsweise das im Sinn der Aufgabenstellung richtige Gesamtergebnis erzielt wird.

Das im Beispiel der Textformatierung (vgl. Kap. 4.1.1) angewendete Verfahren zur Entflechtung von Teilaufgaben verlagert diese Aufgabe in das Programm, welches eine Partitionierung der Gesamtdatenmenge in eine informationskonsistente Folge von Datenblöcken vornimmt. Dieses Partitionierprogramm wird in das Eingabeprogramm integriert, sodaß man von einer Überführung der Eingabefunktion in eine Eingabefunktion mit integrierter Vorverarbeitung (Partitionierung) sprechen könnte.

Eine Funktion dieses Programms sorgt z. B. dafür, daß ein zu formatierender Text in Absätze partitioniert wird, bei denen zu Beginn eines Absatzes der Text nicht eingerückt sein darf (zur Implementierungstechnik siehe Kap. 4.1.1). Nur Absätze, die dieser Forderung genügen, sind bezgl. der Teilaufgabe "Zeilenumbruch mit Randausgleich" entflochten. Absätze, für die dies nicht gilt, erfordern für die Teilaufgabe "Zeilenumbruch mit Randausgleich" Informationen über früher bearbeitete Absätze (die Einrückgröße), die aus der Initialisierung der Teilaufgabe nicht rekonstruierbar ist.

Ein in gewissem Sinn dazu entgegengesetztes Vorgehen wird in [Mes82] bei einem Verfahren zur Parallelisierung von Compilern beschrieben. Dort wird der Datenstrom an beliebigen Stellen geteilt und eigentlich für Teilaufgaben notwendige Informationen bei der Ausführung zunächst offengelassen. Ein in der Ausgabe integriertes Programm zur Nachbearbeitung der Ergebnisse schließt vorher nicht ausfüllbare Lücken.

Abschließend seien die beiden angesprochenen Verfahren zur Entflechtung von Teilaufgaben kurz am Beispiel der Sortierung einer Zahlenfolge nachvollzogen.

Abb. 2.16 zeigt Vor- und Nachbearbeitung bei der Sortierung von Teilketten von Zahlen, bei denen mit der Datenaufteilung eine Art Steuerung durch Vorsortierung in gewisse Bereiche stattfindet. Abb. 2.17 zeigt Vor- und Nachbearbeitung bei der Sortierung von Teilketten von Zahlen, bei denen der Datenstrom ohne Vorinterpretation geteilt wird.

Die Beispiele zeigen, daß eine Verarbeitung mit integrierter Vorinterpretation hier eine Nachbearbeitung bei der Ausgabe der Ergebnisse einspart.

Das Verlassen von Programmfunktionen nach Bearbeitung eines einzelnen Datenblocks kann in der E/A-orientierten Form mit Kontrolle (vgl. Def. 2.7) über die Auswertung einer entsprechenden Prozedurquittung erzwungen werden.

```

MODULE PSORT
  /*DATENAUFTEILUNG MIT VORINTERPRETATION*/

  /*      -----      AUSSENBEZUEGE      -----      */

  VALGLOB SUBR (INT, INT) INT EAKSORT;
  VALGLOB SUBR (INT, REF INT) INT EINGABE;
  VALGLOB SUBR (INT, INT) INT AUSGABE;
  /*wie in Kap. 2.2 deklariert*/

  /*      -----      PSORT      -----      */

  GLOBAL SUBR PSORT = () INT

  BEGIN
    INT STDIN $= ...; /*STANDARD-EINGABEKANAL*/
    INT STDOUT $= ...; /*STANDARD-AUSGABEKANAL*/

    INT QUITTUNG, WERT;

    QUITTUNG := EINGABE(STDIN,WERT);
    WHILE QUITTUNG /= EOF
      DO IF "Wert innerhalb bestimmter Grenzen"
        THEN AUSGABE("spez. Bereichskanal Eingabe", WERT)
        FI;
        QUITTUNG := EINGABE(STDIN,WERT)
      OD;

    FOR I TO "Anzahl verschiedener Bereiche"
      DO EAKSORT("Bereichskanal Eingabe","Bereichskanal Ausgabe")
      OD;

    FOR I TO "Anzahl verschiedener Bereiche"
      DO QUITTUNG := EINGABE("Bereichskanal Ausgabe",WERT);
        WHILE QUITTUNG /= EOF
          DO AUSGABE(STDOUT,WERT);
            QUITTUNG := EINGABE("Bereichskanal Ausgabe",WERT)
          OD
        OD
      OD
    END

  MODEND

```

Abb. 2.16: Aufteilung einer Datenmenge zur Sortierung mit in der Eingabe integrierter Vorinterpretation

```
MODULE SORTN
/*DATENAUFTEILUNG OHNE VORINTERPRETATION*/

/*      -----      AUSSENBEZUEGE      -----      */

VALGLOB SUBR (INT, INT) INT EAKSORT;
VALGLOB SUBR (INT, REF INT) INT EINGABE;
VALGLOB SUBR (INT, INT) INT AUSGABE;
/*wie in Kap. 2.2 deklariert*/

/*      -----      SORTN      -----      */

GLOBAL SUBR SORTN = () INT

BEGIN
  INT STDIN $= ...; /*STANDARD-EINGABEKANAL*/
  INT STDOJT $= ...; /*STANDARD-AUSGABEKANAL*/

  INT QUITTUNG, WERT;

  QUITTUNG := EINGABE(STDIN,WERT);
  WHILE QUITTUNG /= EOF
  DO FOR I TO "Anzahl verschiedener Bereiche"
    WHILE QUITTUNG /= EOF
      DO AUSGABE("Bereichskanal Eingabe",WERT) OD;
      QUITTUNG := EINGABE(STDIN,WERT)
    OD;

  FOR I TO "Anzahl verschiedener Bereiche"
  DO EAKSORT("Bereichskanal Eingabe","Bereichskanal Ausgabe")
  OD;

  "lies von jedem Bereich das Minimum ein";
  WHILE "nicht fertig"
  DO "gib Minimum der Zahlen aus, die zuletzt von den
    verschiedenen Bereichen eingelesen wurden";
    "lies aus dem Bereich eine neue Zahl ein, aus dem
    das letzte ausgegebene Minimum stammte"
  OD
END

MODEND
```

Abb. 2.17: Aufteilung einer Datenmenge zur Sortierung ohne Vorinterpretation bei der Eingabe

In Kap. 4.1 wird am Beispiel der Textformatierung ausführlich auf die hier angerissenen Problemstellungen mit der parallelen Verarbeitung von informationskonsistenten Folgen von Datenblöcken eingegangen.

2.5 Zusammenfassung, Modultypen und Modulhierarchie

Die in den Kapiteln 2.2, 2.3 und 2.4 eingeführten Methoden erlauben nun die Erläuterung der Schritte zur Umformung eines sequentiellen Ausgangsprogramms in ein Netz von Teilaufgaben, die ohne Änderung von Programmstrukturen von Prozessoren eines Multiprozessorsystems ausführbar sind.

Im einzelnen sind dies:

1. Modularisierung nach der Methode des Composite Design (für ordnungserhaltende Programmsysteme normalerweise gegeben)
2. Gestaltung der Schnittstellen eines Moduls zu anderen Modulen so, daß benötigte Eingaben und erzielte Ausgaben über Kommunikationsprozeduren angefordert bzw. abgesetzt werden
3. Definition der Kommunikationsprozeduren in einem eigenen Modul
4. Bestimmung einer minimalen Arbeitsmenge für jede Teilaufgabe
5. Formulierung einer Routine, die den gesamten Datenbestand in eine informationskonsistente Folge von Datenblöcken partitioniert
6. Festlegung einer zulässigen Verarbeitungsstrategie

Die Vorgehensweise und die Festlegung einer Verarbeitungsstrategie resultieren in der in Abb. 2.18 dargestellten Modulhierarchie.

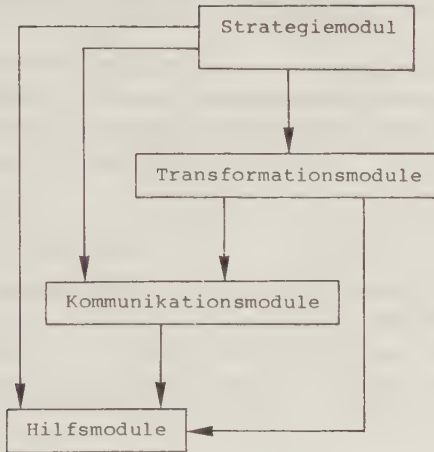


Abb. 2.18: Aufrufhierarchie der Module als Ergebnis der durchgeführten Umformungen

Die Modultypen sind dabei wie folgt definiert:

Def. 2.11: Unter dem Strategiemodul versteht man den Modul, in dem die Verarbeitungsstrategie niedergelegt ist. Er beschreibt im wesentlichen das Zusammenwirken der verschiedenen Transformationsmodule.

Def. 2.12: Transformationsmodule führen die Transformation von Datenmengen durch. Sie werden je nach Verarbeitungsvorschrift und Bearbeitungsstand vom Strategiemodul aufgerufen.

Die Zusammenfassung aller Transformationsmodule entspricht dem ursprünglichen Programm, welches nach den beschriebenen Umformungen in die Modulhierarchie der Abb. 2.18 eingebettet ist.

Def. 2.13: Kommunikationsmodule sind die Module, innerhalb derer die Kommunikationsprozeduren ausgeführt werden. Die Routinen der Kommunikationsmodule können von Strategie- und Transformationsmodulen angesprochen werden.

Def. 2.14: Zu den Hilfsmodulen zählen Pakete aus Programmbibliotheken zur Realisierung bestimmter Funktionen oder zur Implementierung von Informationsspeichern. Routinen dieser Module können von den übrigen Modultypen verwendet werden, rufen jedoch keine Routinen der anderen Modultypen auf.

Innerhalb der Transformationsmodule wird der eigentliche Algorithmus realisiert. Abb. 2.19 zeigt den E/A-orientierten Aufbau eines Transformationsmoduls und das Zusammenwirken mit seiner Umwelt.

Bemerkungen:

1. Nur auf der Ebene der Hilfsmodule läßt sich eine eindeutige Unterscheidung in F(unktions)-Module und D(aten)-Module angeben, wie sie in [Alt78] vorgenommen wird. In den höheren Ebenen sind eher Zwischenformen wie sie [GoKa78] beschreibt, vorherrschend.
2. Die Entflechtung von Teilaufgaben kann als Überführung funktionsorientierter Module mit Gedächtnis in funktionsorientierte Module ohne Gedächtnis aufgefaßt werden.
3. Parallelisierungsverfahren beeinflussen nur den Strategiemodul und die Kommunikationsmodule.

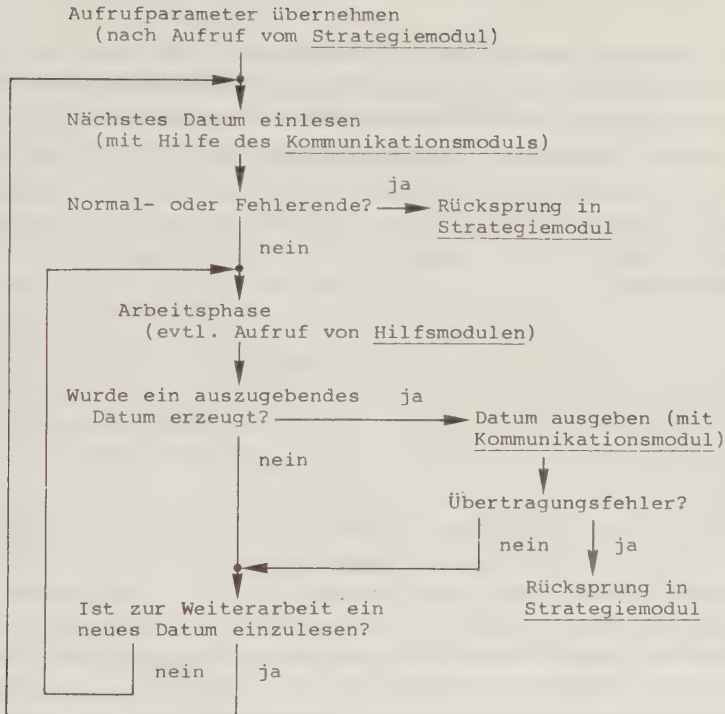


Abb. 2.19: E/A-orientierte Struktur eines Transformationsmoduls und seine Umwelt

In Kap. 3 werden parallele Verarbeitungsstrategien auf der Basis dieser Modulhierarchie zusammengestellt. Kap. 4 gibt Ergebnisse und Bewertungen anhand von Anwendungsbeispielen.

3 Parallele Verarbeitungsstrategien für ordnungserhaltende Programmsysteme

Die folgenden Ausführungen stellen einige Strategien zur Bearbeitung der in Kap. 2 entwickelten Programmstrukturen durch Prozessoren eines Multiprozessorsystems zusammen.

In diesem Rahmen sind zwei Hauptprobleme zu lösen. Das erste betrifft das der Zuordnung von Programmfunktionen zu Prozessoren (vgl. hierzu [From82]), das zweite das der Festlegung von Reihenfolgen in der Bearbeitung der verschiedenen Teilaufgaben (Scheduling).

Kap. 3.1 gibt Anmerkungen für die Zuordnung von Programmfunktionen zu Prozessoren (soweit dies für das zugrundeliegende Multiprozessorsystem möglich ist) und zeigt, daß für diese Art der Aufgabenaufteilung bezgl. der Fragen von Zuordnungen und Bearbeitungsstrategien sehr enge Grenzen gesetzt sind.

Ganz im Gegensatz dazu gibt es für die in Kap. 3.2 beschriebenen Strategien zur parallelen Bearbeitung von Aufgaben, die durch Teilung der Datenmenge entstanden sind, eine Fülle möglicher Ablaufschemata, deren Güte zudem noch in Abhängigkeit der Teilaufgabengröße gesehen werden muß. Ausgehend von einer Betrachtung des Rahmens möglicher Abarbeitungsfolgen und Parametern zur Annäherung an optimale Lösungen werden für hierarchische Strukturen geeignete Heuristiken formuliert.

In Kap. 3.3 werden Probleme der Übertragbarkeit der Heuristiken auf andere Rechnerstrukturen angeschnitten und Vorschläge für "bessere" Rechnerarchitekturen gegeben.

3.1 Die Zuordnung von Programmfunktionen zu Prozessoren eines hierarchischen Multiprozessorsystems

Wenn die Zahl der Teilaufgaben im Programm nicht größer als die Zahl der verfügbaren Prozessoren ist, besteht die Möglichkeit, jede Programmfunktion einem Prozessor zuzuordnen.

Damit ist es nicht erforderlich, Betrachtungen zur Aufteilung der Datenmenge anzustellen. Zu beachten ist lediglich, daß Programmfunktionen, die miteinander Daten austauschen, auf Prozessoren liegen, die auf einen gemeinsamen Speicher zugreifen können oder untereinander Botschaften senden und empfangen können. Diese Forderung, bei einer Kette nacheinander auszuführender Teilaufgaben im allgemeinen problemlos erfüllbar, führt bei Verzweigungen im die Teilaufgaben beschreibenden Ablaufschema zu starken Einschränkungen bei der Wahl möglicher Zuordnungen (vgl. Kap. 4.1).

Ist eine zulässige Zuordnung gefunden (zur Definition und zum Verfahren zur Konstruktion einer zulässigen Zuordnung siehe [From82]), bleibt diese für den gesamten Programmlauf erhalten.

Für den Fall einer Kette von aufeinanderfolgenden Teilaufgaben, bei denen nur in der Kette benachbarte Teilaufgaben miteinander Daten austauschen, sind Pipelining-Verfahren unmittelbar anwendbar. Pipelining-Verfahren sind ein häufig angewendetes Konzept, um Programmlaufzeiten über die Vervielfachung von Arbeitseinheiten zu verkürzen.

Def. 3.1: Pipelining¹⁷⁾ bezogen auf ein bestimmtes Abstraktionsniveau liegt dann vor, wenn die auf diesem Niveau definierten Aktionen in Teilaktionen unterteilt sind, für deren Bearbeitung auf dem vorhandenen Rechensystem spezialisierte Teilwerke bereitstehen, so daß eine Folge von Aktionen im Sinne einer

17) Definition in Anlehnung an [BoHä83].

Fließbandverarbeitung mit überlappter Verarbeitung ausgeführt werden können, zu einem gegebenen Zeitpunkt mehr als eine Aktion - jedoch in verschiedenen Teilaktionen - bearbeitet wird. Die Spezialisierung der Teilwerke der Pipeline muß dabei nicht notwendig fest sein - sie kann durchaus variabel - etwa programmierbar sein.

Abb. 3.1 zeigt das Grundschemata der Pipeline-Verarbeitung anhand eines Gantt-Diagramms.

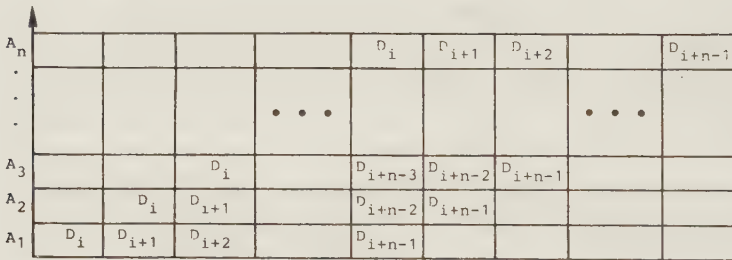


Abb. 3.1: Pipelining-Ausführung einer Aufgabe A auf einer Folge von Datenobjekten D_i

Pipelining wird auf verschiedenen Ebenen angewendet, als Phasenpipelining auf der Ebene des Befehls- bzw. Rechenwerkes (arithmetisches Pipelining), als Befehlspipelining auf der Ebene verschiedener Funktionseinheiten (arithmetisch logische Einheit, E/A-Einheit), als Makropipelining auf der Ebene verschiedener (programmierbarer) Prozessoren.¹⁸⁾ Eine Klassifikation der verschiedenen Arten von Pipelining wird in [Swob75] gegeben. [RaKi74] und [RaLi74] gehen auf arithmetisches und

18) Dementsprechend lassen sich hier untersuchte Aufgaben in die Ebene des Makropipelining einordnen.

Befehlspipelining ein, in [Hän73] wird auf Makropipelining eingegangen, in [RaLi77] werden Charakteristika einiger Pipelining-Rechner zusammengefaßt.

Pipelining-Verfahren sind attraktiv (etwa im Vergleich zur Baumhöhenreduktion), da sie einen Geschwindigkeitsgewinn gegenüber der sequentiellen Verarbeitung erreichbar erscheinen lassen, der der Zahl der Verarbeitungsstufen in der Pipeline entspricht. Eine natürliche Grenze für den erreichbaren Geschwindigkeitsgewinn ergibt sich jedoch daraus, daß es nicht möglich ist, eine Aufgabe in beliebig viele Phasen zu zerlegen.

Sei die Aufgabe A in die n Teilaufgaben A_1 bis A_n zerlegt. Sei die Laufzeit einer Teilaufgabe A_i zur Verarbeitung eines Datums durch $T(A_i)$ gegeben und seien m Daten zu verarbeiten. Dann gilt für die Gesamtausführungszeit $T(A)$ im sequentiellen Fall

$$T_{\text{seq}}(A) = m * \sum_{i=1}^n T(A_i)$$

und für die Pipeline-Verarbeitung¹⁹⁾

$$T_{\text{pip}}(A) \leq (m+n-1) * T(A_i)_{\text{max}}$$

Damit ergibt sich für den Geschwindigkeitsgewinn bei pipelineartiger Verarbeitung, definiert durch $T_{\text{seq}}(A)/T_{\text{pip}}(A)$

$$S_{\text{pip}} \geq \frac{m * \sum_{i=1}^n T(A_i)}{(m+n-1) * T(A_i)_{\text{max}}}$$

¹⁹⁾ Bei einer Pipeline wird die Gesamtverarbeitungszeit durch die Verarbeitungszeit der langsamsten Teilaufgabe bestimmt. Außerdem sind zusätzlich n-1 Verarbeitungsschritte erforderlich, um die Pipeline zu füllen bzw. zu leeren.

Falls alle Teilaufgaben gleiche Bearbeitungszeit haben (Idealfall), ergibt sich

$$S_{pid} = \frac{m \cdot n}{m+n-1}$$

Wenn die Zahl m der zu verarbeitenden Daten sehr hoch ist, ist somit mit Pipeliningverarbeitung unter den oben genannten Voraussetzungen ein nahezu n -facher Geschwindigkeitsgewinn gegenüber der sequentiellen Verarbeitung möglich. Auf der anderen Seite kann eine Teilaufgabe, die signifikant länger dauert als die übrigen Teilaufgaben die Arbeitsweise der Pipeline so verschlechtern, daß gegenüber der sequentiellen Verarbeitung fast kein Geschwindigkeitsgewinn erzielt wird.

Wesentliche Parameter einer Pipeline sind neben der Zahl der Stufen und der Menge der zu verarbeitenden Daten die Größe der Puffer zur Aufnahme von zwischen Teilaufgaben auszutauschenden Daten und die Art ihrer Weitergabe (synchron oder asynchron).²⁰⁾ In [RaKi74] wird gezeigt, daß in Fällen konstanter Länge von Teilaufgaben, bei denen jedes einzelne Eingabedatum zu genau einem Ausgabedatum verarbeitet wird, die Gesamtbearbeitungsdauer unabhängig von der Größe der Zwischenpuffer ist. In [From82] wird abgeleitet, daß in diesem Fall die Gesamtbearbeitungsdauer unabhängig davon ist, ob die Daten synchron oder asynchron weitergegeben werden.

Für ordnungserhaltende Programmsysteme, die nach dem Verfahren des Kap. 2.2 konstruiert wurden, sind Pipelining-Verfahren einfach zu implementieren.

20) Bei der synchronen Weitergabe werden Ergebnisse einer Teilaufgabe erst dann weitergegeben, wenn in der augenblicklichen Stufe alle Teilaufgaben ihr Ergebnis erzeugt haben, bei der asynchronen Weitergabe wird ein Ergebnis weitergegeben, sobald es erzeugt ist.

Nach erfolgter Zuordnung der Teilaufgaben zu den verschiedenen Prozessoren des Systems ist im Strategiemodul statt einer sequentiellen Verarbeitungsstrategie ein Aufruf einzusetzen, der die verschiedenen Funktionen auf den verschiedenen Prozessoren aufruft und statt der Kommunikationsprozeduren zur Ein- und Ausgabe auf Dateien Routinen zu implementieren, die Datenpuffer füllen (Ausgabe) bzw. leeren (Eingabe).

Wenn nur der Prozessor in der Spitze der Hierarchie in der Lage ist, Ein- und Ausgaben durchzuführen, ergeben sich zwei "extreme" Verarbeitungsstrategien. In der eingabeorientierten Strategie werden von diesem Prozessor so lange Daten in die Pipeline gelesen, bis die Puffer der Pipeline gefüllt sind und dann erst Ausgaben durchgeführt. In der ausgabeorientierten Strategie werden so lange anstehende Ausgaben ausgeführt, bis die Pipeline geleert ist und dann erst neue Eingaben in die Pipeline eingebracht. Dazwischen lassen sich Strategien denken, die versuchen, den Belegungsstand der Datenpuffer immer auf einem mittleren Niveau zu halten.

Im allgemeinen wird man ausgabeorientierte Strategien vorziehen, da sie dem Prinzip "Eingaben so spät wie möglich, Ausgaben so früh wie möglich" entsprechen. Dies gilt umso mehr, als eine eingabeorientierte Strategie in einem EGPA-ähnlichen hierarchischen Multiprozessorsystem zu Systemverklemmungen führen kann.

Betrachten wir dazu ein Beispiel. In einem EGPA-System möge eine Pipeline so realisiert sein, daß die vom B-Prozessor eingelesenen Daten nacheinander von den Prozessoren A_1 bis A_4 verarbeitet werden und anschließend vom B-Prozessor wieder ausgegeben werden.

Dabei ergibt sich das Problem, daß fünf Zwischenergebnisse auf den vier Speichern der A-Prozessoren zwischengespeichert werden müssen, d. h. der Speicher eines A-Prozessors zwei verschiedene Zwischenergebnisse aufnehmen muß. Dann kann im Fall ungleich langer Zeiten unter den Teilaufgaben zur Verarbeitung eines Datums der Speicher, auf dem zwei verschiedene Zwischenergebnisse abgelegt werden sollen, vollständig mit einem Zwischenergebnis-

Auch wenn für Programme auf der Basis einer pipelineartigen Verarbeitung der Raum möglicher Verarbeitungsstrategien sehr stark eingeschränkt ist und in der Regel wegen ungleich langer Laufzeiten von Teilaufgaben nur geringe Geschwindigkeitsgewinne erwarten läßt (vgl. Kap. 4.1), haben diese doch ihre Berechtigung. Dadurch, daß hier Einflüsse wie Größe von Puffern zur Aufnahme von Zwischenergebnissen bzw. synchrone oder asynchrone Weitergabeform von Zwischenergebnissen praktisch keine Rolle spielen und die Länge von Teilaufgaben nicht beeinflußbar ist (wie etwa in den Strategien des Kap. 3.2), können Einflüsse organisatorischer Art wie z. B. verschiedene Formen von Synchronisation recht gut quantitativ beurteilt werden, ohne daß hierzu eine Modellierung auf der Ebene einzelner Instruktionen erforderlich wäre (vgl. hierzu Kap. 5.2).

Obige Aussagen bezgl. der Größe von Zwischenpuffern und Einflüssen synchroner und asynchroner Weitergabe von Daten auf Programmlaufzeiten lassen sich in der Form nicht mehr aufrecht erhalten, wenn der Aufgabengraph Verzweigungen enthält oder mehrere Eingaben angefordert werden, bevor eine Ausgabe entsteht bzw. aus einer Eingabe mehr als eine Ausgabe entsteht. Gleiches gilt, wenn Laufzeiten von Teilaufgaben zwischen Eingabeanforderung und Ausgabeerzeugung stark streuen. Im Fall von Verzweigungen im Aufgabengraphen kann sich die Notwendigkeit ergeben, daß Puffer eine bestimmte Größe haben müssen, um Verklemmungsfreiheit zu garantieren (vgl. hierzu Kap. 4.1.1).

Verfahren, die Programmfunktionen Prozessoren eines Multiprozessorsystems zuordnen, spielen jedoch hauptsächlich aus dem Grund eine untergeordnete Rolle, weil sie aufgrund unterschiedlich langer Bearbeitungszeiten der verschiedenen Teilaufgaben das System schlecht auslasten. Man versucht dann, diese Teilaufgaben in weitere pipelineartig ausführbare Teile zu unterteilen oder die Teilaufgaben von mehr als einem Prozessor ausführen zu lassen. Um in letzterem Fall einen Geschwindigkeitsgewinn zu erzielen, werden die von dieser Teilaufgabe zu verarbeitenden Daten aufgeteilt. Dies führt auf die in Kap. 2.3

angestellten Überlegungen zur Aufteilung der zu verarbeitenden Datenmenge auf die Prozessoren des Systems und die in Kap. 3.2 auf dieser Grundlage aufbauenden Verfahren zur Parallelverarbeitung.

3.2 Die Aufteilung der Datenmenge auf Prozessoren

Nachstehend beschriebene Verfahren zur parallelen Verarbeitungsordnungserhaltender Programmsysteme basieren auf der in den Kapiteln 2.3 bzw. 2.4 entwickelten und in Abb. 2.15 dargestellten Aufgabenstruktur. Es wird vorausgesetzt, daß die zu verarbeitende Datenmenge in eine informationskonsistente Folge von Datenblöcken partitioniert und bezgl. der zwischen "Eingabe" und "Ausgabe" liegenden Teilaufgaben eine Entflechtung dieser Blöcke durchgeführt wurde.

Dann bietet sich für die Parallelverarbeitung auf einem hierarchischen Multiprozessorsystem folgende Arbeitsweise an:

1. Der Prozessor in der Spitze führt die Teilaufgaben "Eingabe" (obere waagrechte Reihe der Abb. 2.15) und "Ausgabe" (untere waagrechte Reihe der Abb. 2.15) aus.
2. Die übrigen einen Datenblock transformierenden Teilaufgaben werden zu einer Teilaufgabe "Verarbeitung" zusammengefaßt und vom Prozessor in der Spitze der Hierarchie zur Bearbeitung an untergeordnete Prozessoren delegiert.
3. Zwischenergebnisse werden mittels der Kommunikationsprozeduren auf den Speichern der betreffenden Prozessoren verwaltet. Informationen, die zwischen dem Prozessor in der Spitze und den übrigen Prozessoren ausgetauscht werden müssen (partitionierte Eingaben, vorverarbeitete Ergebnisse) werden mittels der Kommunikationsprozeduren in beiden Prozessortypen zugreifbaren Speichern (Kommunikationsspeichern) bzw. als Bestandteil von zwischen den Prozessoren zu übermittelnden Botschaften niedergelegt.

Im folgenden werden der Prozessor in der Spitze der Hierarchie als Kontrollprozessor und die übrigen Prozessoren als Arbeitsprozessoren bezeichnet.

Für die Teilaufgaben zur Transformation des Datenblockes D_i werden die Bezeichnungen E_i für die Eingabe, V_i für die Verarbeitung und A_i für die Ausgabe von D_i verwendet. Abb. 3.3 zeigt ein Beispiel eines Graphen für die Transformation einer informationskonsistenten Folge aus vier entflochtenen Datenblöcken.

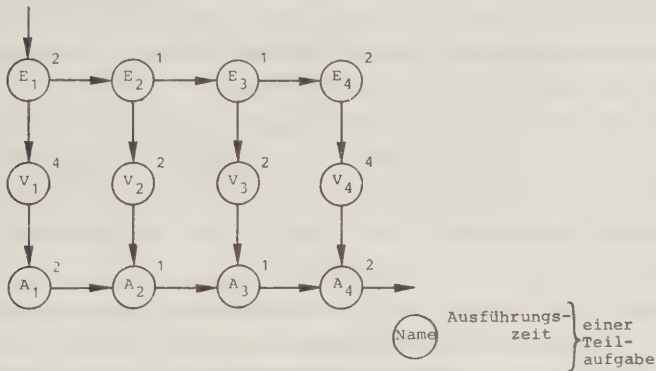


Abb. 3.3: Beispiel eines Aufgabengraphen für ein ordnungserhaltendes Programmsystem mit einer informationskonsistenten Folge aus vier entflochtenen Datenblöcken

Das den Beschreibungen paralleler Verarbeitungsstrategien zugrundeliegende hierarchische Multiprozessorsystem besteht aus einer zweistufigen Hierarchie mit einem Kontrollprozessor und p mit diesem verbundenen Arbeitsprozessoren.²¹⁾ Die Überlegungen zur parallelen Verarbeitung ordnungserhaltender Programmsysteme auf dieser Struktur lassen sich jedoch direkt auf mehrstufige Hierarchien übertragen, wenn die Prozesselemente der Zwischenschichten als Transporteure von Informationen zwischen dem Kontrollprozessor und den Arbeitsprozessoren der unteren Schicht betrachtet werden.

3.2.1 Der Raum möglicher Lösungen, optimale Lösungen

Abgesehen von Fragen der Zuordnung von zu verarbeitenden Datenblöcken zu Prozessoren tieferer Schichten der Hierarchie (Arbeitsprozessoren) ergibt sich bei der Bearbeitung der Aufgaben auf dem Prozessor in der Spitze (Kontrollprozessor) das Problem zu entscheiden, in welcher Reihenfolge Ein- und Ausgaben zu erfolgen haben.

Im Rahmen dieser Betrachtungen kann bezgl. der Zuordnung von zu verarbeitenden Datenblöcken zu Arbeitsprozessoren und bezgl. der Festlegung der Reihenfolge, in der Ein- und Ausgaben durchgeführt werden, in zwei Richtungen optimiert werden.

1. Es soll ein Minimum an Speicher zur Aufnahme von zwischen Prozessoren auszutauschender Information benötigt werden ("eingeschränkte Ressourcen").
2. Die Gesamtausführungszeit soll bezgl. des zugrundeliegenden Multiprozessorsystems minimal werden.

Eine Strategie, die bezgl. des zu verarbeitenden Aufgabengraphen in einem hierarchischen Multiprozessorsystem mit einem Kontrollprozessor und p Arbeitsprozessoren die erste Forderung erfüllt, wird in Anlehnung an das Prinzip des "alternating sequential/parallel processing" [Wal82] konstruiert.

Strategie 3.2.1.1:

Der Kontrollprozessor liest auf jeden Arbeitsprozessor einen Datenblock. Die eingelesenen Datenblöcke werden von den Arbeitsprozessoren parallel weiter verarbeitet. Anschließend gibt der Kontrollprozessor reihenfolgerichtig die von den

- 21) Die gleiche Architektur wird in [Wal82] für parallele Verarbeitungsstrategien zugrundegelegt, die auf dem Prinzip des "Alternating sequential/parallel processing" beruhen. Diese bilden eine Grundlage der hier skizzierten Verarbeitungsstrategien (vgl. Kap. 3.2.1).

Arbeitsprozessoren erzielten Ergebnisse aus.

Dieser Zyklus wird so lange wiederholt, bis der gesamte Datenbestand verarbeitet wurde.

Für die Betrachtungen zur Minimierung des Bedarfs an Kommunikationsspeicher ist es dabei belanglos, ob die Arbeitsprozessoren asynchron (d. h. unmittelbar nach Erhalt des zu verarbeitenden Datenblocks) oder synchron (d. h. erst dann, wenn sämtliche Arbeitsprozessoren einen Datenblock zur Verarbeitung haben) gestartet werden.

Eine streng synchrone Variante des "Alternating Sequential Parallel Processing" ist im SMS-System [KKK76] realisiert. Dort wird mit der Arbeitsphase (autonomen Phase) erst dann begonnen, wenn alle Arbeitsprozessoren "ihre" Eingabe haben und Ergebnisse und neue Eingaben erst dann ausgetauscht (Kommunikations- und Kontrollphase), wenn alle Arbeitsprozessoren ein Ergebnis angeliefert haben.

Im Fall der asynchronen Arbeitsweise der Arbeitsprozessoren ergeben sich zwei Ebenen von Parallelismus. Zum einen arbeiten die Arbeitsprozessoren untereinander parallel (im Sinne von "nebenläufig"), zum anderen ergibt sich Parallelarbeit zwischen Kontroll- und Arbeitsprozessoren (im Sinne von "pipelineartig"). In [Vas84] wird diese häufig im EGPA-System realisierte Form der Parallelverarbeitung in Anlehnung an den in [Wal82] eingeführten Begriff ASP ("alternating sequential/parallel processing") als APP ("alternating parallel/parallel processing") bezeichnet.

Die Strategie 3.2.1.1 minimiert zwar den Bedarf an Speicher zur Kommunikation zwischen Kontrollprozessor und Arbeitsprozessoren. Sie wird jedoch nur sehr selten bezüglich der Minimierung der Gesamtausführungszeit optimal sein, da nur wenig wirkliche Parallelarbeit der verschiedenen Prozessoren stattfindet.

Wenn Speicher zur Aufnahme von zwischen Prozessoren auszutauschender Information in ausreichender Menge²²⁾ vorhanden ist, lassen sich folgende Überlegungen zur Minimierung der Gesamtausführungszeit anwenden:

1. Eine Minimierung der Gesamtausführungszeit im betrachteten hierarchischen Multiprozessorsystem ergibt sich dann, wenn mit der Verarbeitung des letzten Datenblocks zum frühest möglichen Zeitpunkt begonnen wird. Da dies aufgrund der Abhängigkeiten zwischen den verschiedenen Teilaufgaben erst dann geschehen kann, wenn sämtliche Eingaben erfolgt sind, ist es zum Erzielen einer möglichst geringen Gesamtausführungszeit sinnvoll, auf dem Kontrollprozessor zuerst sämtliche Eingaben durchzuführen, bevor das erste Ergebnis ausgegeben wird.
2. Eine Minimierung der Gesamtausführungszeit im betrachteten hierarchischen Multiprozessorsystem ergibt sich vermutlich dann, wenn mit der Verarbeitung eines Datenblocks durch einen Arbeitsprozessor unmittelbar dann begonnen werden kann, wenn dieser eingelesen wurde, d. h. der Arbeitsprozessor sollte zu diesem Zeitpunkt möglichst nicht mit anderer Arbeit belastet sein. Man sollte also einen Datenblock einem Arbeitsprozessor zur Verarbeitung zuordnen, der momentan nichts oder "möglichst wenig" zu tun hat. (Daß diese Überlegung nicht in jedem Fall zur Lösung mit minimaler Gesamtausführungszeit führt, wird am Beispiel 3.2.1.3 deutlich.)

Intuitiv wird man dabei von der Idealannahme ausgehen, daß die Zeit zur Verarbeitung eines Datenblocks durch einen Arbeitsprozessor für alle Datenblöcke nahezu gleich ist und daraufhin vermuten, daß die Gesamtausführungszeit minimal wird, wenn Datenblöcke "round robin" an die Arbeitsprozessoren verteilt werden.

22) Damit soll gemeint sein, daß der Kommunikationsspeicher groß genug ist, die gesamte Menge an zwischen Prozessoren auszutauschender Information aufzunehmen.

Diese beiden Überlegungen führen auf die Strategie 3.2.1.2.

Strategie 3.2.1.2:

Der Kontrollprozessor liest den gesamten Datenbestand auf die Speicher der Arbeitsprozessoren, wobei einzelne Datenblöcke im Round-Robin-Verfahren auf die Arbeitsprozessoren verteilt und dort gepuffert werden. Die Arbeitsprozessoren durchlaufen einen Zyklus "Falls Eingabepuffer leer, warte auf einen neuen Datenblock; verarbeite den nächsten Datenblock", bis alle Datenblöcke verarbeitet worden sind. Erst wenn der gesamte Datenbestand eingelesen ist, erfolgt die Ausgabe (mit evtl. notwendiger Nachbearbeitung) der durch die Arbeitsprozessoren verarbeiteten Datenblöcke.

Daß diese Strategie unter bestimmten (idealisierenden) Voraussetzungen wirklich zu einer Lösung mit minimaler Gesamtausführungszeit führt, zeigt Satz 3.2.

Satz 3.2: Sei der durch ein ordnungserhaltendes Programmsystem zu verarbeitende Datenbestand in eine informationskonsistente Folge von Datenblöcken partitioniert. Seien in einem hierarchischen Multiprozessorsystem mit einem Kontrollprozessor und p mit ihm verbundenen Arbeitsprozessoren die (bezgl. der Datenblöcke sequentiell durchzuführenden) Teilaufgaben "Eingabe" und "Ausgabe" nur auf dem Kontrollprozessor und die (bezgl. der Datenblöcke parallel ausführbaren) Teilaufgaben "Verarbeitung" nur auf den Arbeitsprozessoren ausführbar. Seien die Ausführungszeiten für die Teilaufgabe "Eingabe" für alle Datenblöcke konstant und die Ausführungszeiten für die Teilaufgabe "Verarbeitung" für alle Datenblöcke konstant. Seien ferner keine Zeiten für Aufgaben organisatorischer Art zu berücksichtigen (vgl. Charakteristik eines idealen Rechensystems in Kap. 2.1.2).

Dann ist die Strategie 3.2.1.2 bezgl. der Minimierung der Gesamtausführungszeit optimal, wenn sie aufgrund der verfügbaren Speicherkapazitäten ausführbar ist.

Beweis: Es wird gezeigt, daß es keine Strategie gibt, die eine kürzere Gesamtausführungszeit liefert.

Seien m Datenblöcke zu transformieren. Sei mit $T(E_i)$ die Dauer der Eingabe für Datenblock i , mit $T(V_i)$ die Dauer der Verarbeitung für Datenblock i und mit $T(A_i)$ die Dauer der Ausgabe für Datenblock i gegeben. Seien mit $F(E_i)$, $F(V_i)$ und $F(A_i)$ die Fertigstellungszeitpunkte der Teile Eingabe, Verarbeitung und Ausgabe der Teilaufgabe i (Transformation von Datenblock i) bezeichnet (vgl. Beispiel der Abb. 3.3). Gemäß Voraussetzung gibt es dann eine Konstante T_E und eine Konstante T_V , sodaß gilt:

$$T_E = T(E_1) = T(E_2) = \dots = T(E_m) \text{ und}$$

$$T_V = T(V_1) = T(V_2) = \dots = T(V_m).$$

Gemäß der Struktur des zu bearbeitenden Aufgabengraphen (vgl. Beispiel der Abb. 3.3) muß ein i_0 existieren, sodaß gilt:

$$F(A_m) = F(V_{i_0}) + \sum_{\mu \leq i_0}^m T(A_\mu)$$

Die Anwendung dieser Beziehung (mit der übrigens der kritische Pfad des Aufgabengraphen beschrieben wird) erlaubt es, im folgenden ohne Beschränkung der Allgemeinheit Zeitbetrachtungen bezgl. der Teilaufgaben "Ausgabe" zu vernachlässigen.

Ein "schnellerer" Algorithmus müßte also für V_{i_0} bei gleichen Größen T_E und T_V für die Ausführungszeiten der Teilaufgaben "Eingabe eines Datenblockes" und "Verarbeitung eines Datenblockes" einen früheren Fertigstellungszeitpunkt $F'(V_{i_0})$ liefern.

Zur weiteren Analyse sind zwei Fälle zu unterscheiden:

1. Die Teilaufgabe V_{i_0} kann unmittelbar dann begonnen werden, wenn der Datenblock i_0 eingelesen wurde, d. h. es gilt $p \cdot T_E \geq T_V$ (vgl. Abb. 3.4).
2. Die Teilaufgabe V_{i_0} kann nicht unmittelbar nach Einlesen des Datenblocks i_0 begonnen werden, da der mit V_{i_0} beauftragte Arbeitsprozessor zu diesem Zeitpunkt noch mit der Verarbeitung eines anderen Datenblocks beschäftigt ist, d. h. es gilt $p \cdot T_E < T_V$ (vgl. Abb. 3.5).

Im Fall 1 muß bezgl. des Fertigstellungszeitpunktes $F'(V_{i_0})$ einer besseren Strategie als der Strategie 3.2.1.2 unter den angenommenen Voraussetzungen gelten:

$$F'(V_{i_0}) < i_0 \cdot T_E + T_V.$$

Nun gilt jedoch aufgrund der Abhängigkeiten bezgl. der Teilaufgaben im zu verarbeitenden Aufgabengraphen:

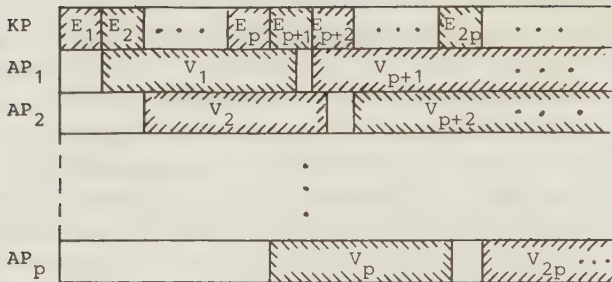
$$F'(V_{i_0}) \geq i_0 \cdot T_E + T_V.$$

Es ergibt sich also ein Widerspruch.²³⁾

Im Fall 2 muß bezgl. des Fertigstellungszeitpunktes $F'(V_{i_0})$ einer besseren Strategie als der Strategie 3.2.1.2 unter den angenommenen Voraussetzungen gelten:

$$\begin{aligned} F'(V_{i_0}) &< F(V_{i_0-p}) + T_V = \\ &= F(V_{i_0-2p}) + 2 \cdot T_V = \dots = \\ &= F(V_{i_0-kp}) + k \cdot T_V. \end{aligned}$$

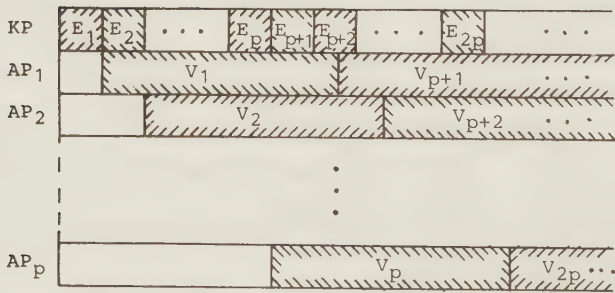
23) Intuitiv ist ohnehin klar, daß in Fällen, in denen unmittelbar nach Einlesen eines Datenblocks eine weitere Verarbeitung dieses Datenblocks stattfinden kann, sicher die Strategie die Gesamtausführungszeit minimiert, bei der die Eingaben zum frühest möglichen Zeitpunkt abgeschlossen werden.



KP: Kontrollprozessor

AP_i : Arbeitsprozessor i

Abb. 3.4: Beweis zum Satz 3.2 - Ablauf nach Strategie 3.2.1.2 Fall 1



KP: Kontrollprozessor

AP_i : Arbeitsprozessor

Abb. 3.5: Beweis zum Satz 3.2 - Ablauf nach Strategie 3.2.1.2 Fall 2

Sei nun k so gewählt, daß $1 \leq (i_0 - kp) \leq p$, d. h. V_{i_0-kp} ist die erste ausgeführte Teilaufgabe "Verarbeitung" des Prozessors, der auch V_{i_0} ausführt. Dann ergibt sich als Forderung an eine bessere Strategie als die Strategie 3.2.1.2:

$$\begin{aligned} F'(V_{i_0}) &< F(V_{i_0-kp}) + k \cdot T_V = \\ &= (i_0 - kp) \cdot T_E + T_V + k \cdot T_V \end{aligned}$$

Die rechte Seite der zuletzt hergeleiteten Ungleichung läßt sich jedoch unter den Voraussetzungen des Satzes 3.2 nur verkleinern, wenn T_E und/oder T_V verkleinert werden im Widerspruch zur Voraussetzung nicht veränderbarer Größen T_E und T_V .²⁴⁾ ■

Die Strategie 3.2.1.2 ist jedoch bezgl. einer Minimierung der Gesamtausführungszeit nicht optimal, wenn die Teilaufgaben ungleich lang sind. Dazu wollen wir folgendes Beispiel betrachten:

Beispiel 3.2.1.3:

Betrachtetes hierarchisches Multiprozessorsystem möge aus einem Kontrollprozessor und zwei Arbeitsprozessoren bestehen und es seien drei Teilaufgaben zu bearbeiten, für deren Bearbeitungszeiten folgende Beziehungen gelten.

$$T(E_1) = T(E_3) = 1; \quad T(E_2) = 2;$$

$$T(V_1) = T(V_3) = 4; \quad T(V_2) = 8;$$

24) Eine weitere (aufgrund der Voraussetzungen von Satz 3.2 ebenfalls nicht zugelassene) Möglichkeit, im Fall 2 die Gesamtbearbeitungsdauer zu verkleinern, besteht darin, den Kontrollprozessor in die Teilaufgaben "Verarbeitung" mit einzubeziehen (vgl. Kap. 3.2.4 und Kap. 4.2.2). In [Wal82] wird in diesem Zusammenhang vorgeschlagen, in einem "Parallelabschnitt" dem Kontrollprozessor die Teilaufgabe mit der kürzesten Verarbeitungszeit zuzuordnen.

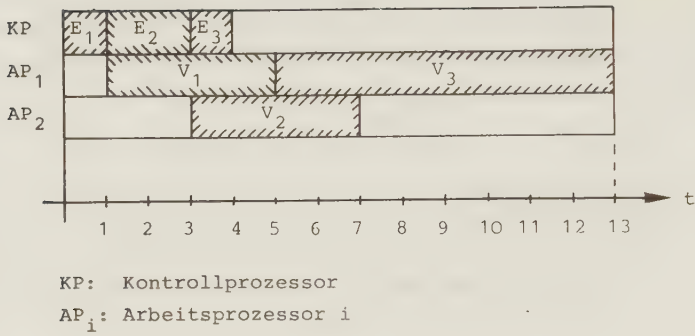


Abb. 3.6: Beispiel 3.2.1.3 - Ablauf nach Strategie 3.2.1.2

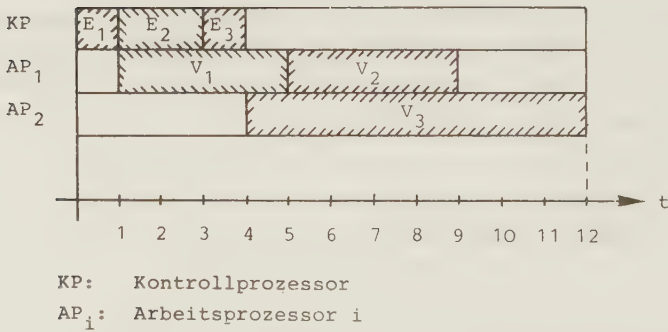


Abb. 3.7: Beispiel 3.2.1.3 - Ablauf nach optimaler Strategie

Die Anwendung der Strategie 3.2.1.2 würde wie in Abb. 3.6 dargestellt einen Fertigstellungszeitpunkt für V_3 von 13 liefern, während die in Abb. 3.7 dargestellte Strategie einen Fertigstellungszeitpunkt für V_3 von 12 liefert.

In [RaChG72] findet sich ein weiteres Beispiel dafür, daß es im Sinne einer Minimierung der Gesamtausführungszeit eines Aufgabengraphen optimal sein kann, einen untätigen Prozessor trotz vorhandener Aufgaben leer stehen zu lassen.

Auf der anderen Seite ist unter den obigen Voraussetzungen die Strategie 3.2.1.2 bezgl. einer Minimierung der Gesamtausführungszeit nicht die einzige Strategie. In obigem Fall liefern alle Strategien die minimale Gesamtausführungszeit, die ein Fertigstellen von V_1 zum gleichen Zeitpunkt wie bei der Strategie 3.2.1.2 ermöglichen oder aufgrund der Zeitverhältnisse innerhalb der einzelnen Teilaufgaben dafür sorgen, daß auf dem Kontrollprozessor zu keinem Zeitpunkt auf das Ergebnis eines Arbeitsprozessors gewartet werden muß.²⁵⁾ Dies umfaßt auch Strategien, bei denen vor Einlesen aller Eingabedaten schon Ausgaben erzielter Ergebnisse stattfinden können. Solche Strategien sind für reale Implementierungen von größerem Interesse, da sie weniger Anforderung an Speicherplatz als die Strategie 3.2.1.2 stellen.

Wenn von weiteren idealisierenden Annahmen, wie sie in Satz 3.2 formuliert sind, abgegangen wird, ist es praktisch nicht mehr möglich, die Zeitverhältnisse bei der Bearbeitung von Teilaufgaben im Aufgabengraphen zu erfassen und die Gesamtausführungszeit über eine Formel auszurechnen. Dann müssen zum Auffinden einer Strategie mit minimaler Gesamtausführungszeit für ein bestimmtes ordnungserhaltendes Programmsystem, welches einen

25) Die Tatsache, daß unter bestimmten Voraussetzungen bezgl. der Laufzeiten verschiedener Teilaufgaben in Strategien mit minimaler Gesamtausführungszeit der Kontrollprozessor nicht auf einen Arbeitsprozessor warten muß, läßt sich experimentell nachprüfen!

bestimmten Datensatz transformiert, prinzipiell alle möglichen Strategien getestet werden. Wie mit Methoden der Kombinatorik abgeleitet werden kann, sind dies bei m zu transformierenden Datenblöcken in einem hierarchischen Multiprozessorsystem mit einem Kontrollprozessor und p Arbeitsprozessoren bei der zu Beginn des Kap. 3.2 skizzierten Arbeitsweise $p^m \cdot \left(\binom{2m}{m} - \binom{2m}{m+1} \right)$ mögliche Alternativen.²⁶⁾ Wenn also keine andere Möglichkeit besteht, eine Strategie mit minimaler Gesamtausführungszeit zu bestimmen, ist das so gestellte Problem - ein Spezialfall des allgemeinen Multiprozessor-Scheduling-Problems - NP-vollständig.

Darüberhinaus gibt es eine Reihe anderer Optimierungskriterien, z. B. Bestimmung der Mindestzahl von Prozessoren, einen gegebenen Aufgabengraphen in Minimalzeit auszuführen oder zur Bestimmung, ob ein Aufgabengraph mit p Prozessoren in der Mindestzeit ausgeführt werden kann [RaChG72].

Zudem muß zur Ermittlung einer optimalen Strategie davon ausgegangen werden, daß die Ausführungszeiten der einzelnen Teilaufgaben a priori bekannt sind, da sonst keine Optimalität bewiesen werden kann [Gon77]. Für die hier untersuchten ordnungserhaltenden Programmsysteme ist dies nicht gewährleistet, da die Ausführungszeit von Teilaufgaben von den Eingabedaten abhängig ist, d. h. es ist für diesen Fall keine optimale Strategie bekannt.

Die Tatsache, daß für die untersuchten (ordnungserhaltenden) Programmsysteme im allgemeinen keine Aussagen über Laufzeiten von Teilaufgaben gemacht werden können, verbietet auch die Anwendung der in [Hoppe84] beschriebenen Verfahren zum deterministischen Zuordnen von Aufgaben zu Prozessoren eines Multiprozessorsystems.

26) Jede von m Eingaben _{m} kann einem von p Arbeitsprozessoren zugeordnet werden (p^m Alternativen); m Eingaben und m Ausgaben können in allen Folgen verarbeitet werden, bei der zu jedem Zeitpunkt die Zahl durchgeführter Eingaben größer oder gleich der Zahl durchgeführter Ausgaben ist ($\left(\binom{2m}{m} - \binom{2m}{m+1} \right)$ Alternativen).

Effiziente Verfahren zum Auffinden optimaler Strategien lassen sich nur für Spezialfälle angeben, z. B. bestimmte reguläre Aufgabenstrukturen, gleiche Ausführungszeiten von Teilaufgaben oder eine sehr eingeschränkte Zahl von Prozessoren. In [From82] finden sich weitere Angaben und Verweise auf die umfangreiche Literatur zum Multiprozessor-Scheduling-Problem.

Für allgemeine Fälle werden deshalb Heuristiken angewendet wie z. B. "keinen Prozessor leer laufen lassen" oder "wenn sich zwei Aufgaben um einen Prozessor bewerben, diejenige aussuchen, die aufgrund der Abhängigkeitsstruktur im zugrundeliegenden Aufgabengraphen früher begonnen werden kann". In Fällen, in denen die Optimalität einer Strategie bewiesen werden kann, liefern solche Heuristiken sehr häufig optimale oder fast optimale Lösungen (vgl. [Hoppe84]).

Ein grundsätzliches Kriterium, welches für viele effiziente Verfahren zur Konstruktion optimaler Strategien Voraussetzung ist, läßt sich in ordnungserhaltenden Programmsystemen über Datenteilung steuern, nämlich zu versuchen, allen Teilaufgaben eines Verarbeitungsschrittes (Eingabe, Umformung, Ausgabe) fast gleiche Verarbeitungszeiten zu geben. Dies kann im allgemeinen erreicht werden, indem verschiedene Datenblöcke zu Bündeln möglichst gleicher Größe vereinigt werden.

Wenn es dadurch gelingt, bei Anwendung der Strategie 3.2.1.2 die in den Abb. 3.4 oder 3.5 dargestellten zeitlichen Reihenfolgen der Fertigstellungszeitpunkte der Teilaufgaben "Eingabe" und "Verarbeitung" zu erhalten, ist Strategie 3.2.1.2 optimal.

Nachfolgend beschriebene Heuristiken gehen von Datenblöcken aus, bei denen die Ausführungszeiten des gleichen Transformations-schrittes für alle Datenblöcke annähernd gleich groß sind. Sie zeigen Möglichkeiten auf, im zugrundeliegenden Multiprozessorsystem auf experimentellem Wege eine parallele Verarbeitungsstrategie zu finden, die von den vorhandenen Ressourcen (Prozessoren, Speichern) ausgehend in einer möglichst geringen Gesamtausführungszeit resultiert. In Kap. 4 werden an zwei Anwendungsbeispielen Ergebnisse dieser Heuristiken dargestellt.

In [Klei82] werden verkehrstheoretische Hilfsmittel und numerische Verfahren bereitgestellt, im voraus Abschätzungen für Gesamtausführungszeiten von nach bestimmten Strategien erstellten Parallelprogrammen anzugeben.

3.2.2 Eine Round-Robin-Heuristik

Bei der Analyse des parallelen Ablaufs bei der Bearbeitung von Teilaufgaben eines ordnungserhaltenden Programmsystems, wie sie durch den Graphen der Abb. 2.15 dargestellt werden, lassen sich drei Zustände unterscheiden:

1. Es liegt noch kein transformierter Datenblock zur Ausgabe vor (hier sind von der Seite des Kontrollprozessors nur Eingaben möglich)
2. Es liegen transformierte Datenblöcke zur Ausgabe vor, und es sind noch Datenblöcke zur Verarbeitung einzulesen (hier sind von der Seite des Kontrollprozessors Eingaben oder Ausgaben möglich)
3. Der gesamte Datenbestand ist eingelesen (hier sind von der Seite des Kontrollprozessors nur noch Ausgaben möglich)

Im folgenden sollen diese drei Zustände als Einschwingzustand, Kernzustand und Auslaufzustand bezeichnet werden.

Während der Ablauf im Auslaufzustand für alle möglichen Verarbeitungsstrategien fest liegt ("Warten, bis ausgabebereites Ergebnis vorliegt; ausgabebereites Ergebnis ausgeben"), sind in den beiden anderen Zuständen Entscheidungen bezgl. des Arbeitsablaufs zu treffen. Im Einschwingzustand ist zu entscheiden, welchem Arbeitsprozessor der nächste einzugebende Datenblock zur Weiterverarbeitung zuzuordnen ist, im Kernzustand²⁷⁾ darüberhinaus, in welcher Abfolge Eingaben und Ausgaben durchgeführt werden sollen.

27) Im Sinne der Warteschlangentheorie kann der Kernzustand als eine Art stationärer Zustand aufgefaßt werden.

Die Round-Robin-Heuristik unterscheidet in Anlehnung an die drei Bearbeitungszustände drei Phasen (vgl. Abb. 3.9):

Die Phase 1 ist die sog. Anlaufphase. In dieser werden (wie in der Strategie 3.2.1.2 des Kap. 3.2.1) so lange Datenblöcke auf die Arbeitsprozessoren geschrieben, bis deren Speicher "gefüllt"²⁸⁾ sind oder der gesamte Datenbestand eingelesen wurde.

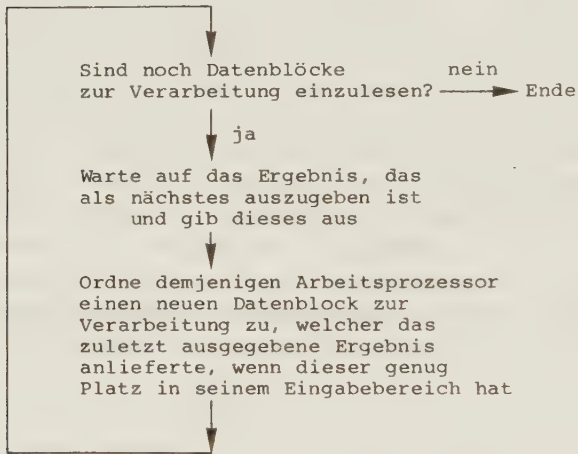
Die Phase 2, die sog. Kernphase beginnt nach Abschluß der Anlaufphase dann, wenn die Eingabedaten noch nicht vollständig eingelesen wurden.²⁹⁾ In dieser wird zunächst auf den Arbeitsprozessor gewartet, von dem das nächste ausgabebereite Ergebnis zu kommen hat. Dieses Ergebnis wird ausgegeben. Wenn (bedingt durch die Ausgabe) der Speicher dieses Prozessors wieder einen Datenblock aufnehmen kann³⁰⁾, wird diesem Prozessor ein neuer Datenblock zur Verarbeitung zugeteilt. Bedingt durch die Verteilung zu verarbeitender Datenblöcke in der Anlaufphase ergibt sich dieser Verlauf für die Arbeitsprozessoren in einer Round-Robin-Schleife so lange, bis der letzte Datenblock eingelesen wurde.

28) Da von vornherein nicht bekannt ist, wieviele Datenblöcke der Eingabespeicher eines Arbeitsprozessors aufnehmen kann, sollte dieser Wert von außen eingegeben werden können.

29) Einschwingzustand und Anlaufphase sind im allgemeinen nicht identisch, da in der Anlaufphase im Gegensatz zum Einschwingzustand auf den Arbeitsprozessoren ausgabefähige Ergebnisse entstehen können. Kernzustand und Kernphase sind im allgemeinen nicht identisch, da am Beginn der Kernphase im Gegensatz zum Kernzustand noch kein Ergebnis zur Ausgabe anstehen muß (vgl. Abb. 3.9).

30) Wie eine solche Abfrage zu implementieren ist, muß in Abhängigkeit der Anwendung und der Form, in der der Kommunikationsspeicher implementiert ist, entschieden werden. Ohne eine solche Abfrage drohen durch Pufferüberlauf - je nach der Art, wie auf dieses Ereignis reagiert wird ("ignorieren" oder "warten, bis genug Platz frei ist") - Datenverlust oder eine Systemverklemmung.

Abb. 3.8 zeigt anhand eines Flußdiagramms die Struktur der Kernphase der Round-Robin-Heuristik. Abb. 3.9 zeigt die Verarbeitung des Programmsystems der Abb. 3.3 mit der Round-Robin-Heuristik durch einen Kontrollprozessor und zwei Arbeitsprozessoren.



Die Phase 3, die sog. Auslaufphase beginnt, wenn der gesamte Bestand an Eingabedaten eingelesen wurde. Dann sind nur noch Ausgaben möglich, wobei aufgrund der Ordnungserhaltung zwischen Eingaben und Ausgaben jedes Mal genau festliegt, von welchem Prozessor das nächste auszugebende Ergebnis zu kommen hat. Hier bleibt nichts anderes übrig, als vor jeder Ausgabe vom Kontrollprozessor aus auf das Fertigwerden dieses Ergebnisses zu warten.

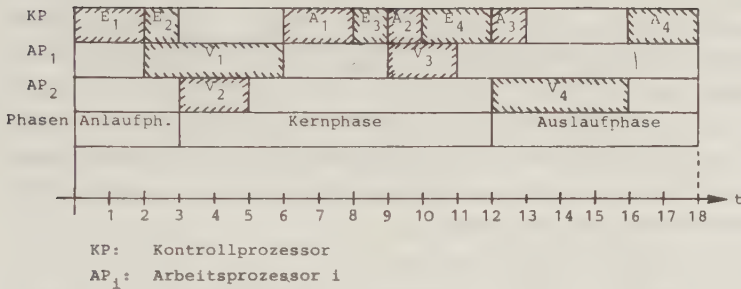


Abb. 3.9: Verarbeitung des Aufgabengraphen der Abb. 3.3 mit der Round-Robin-Heuristik durch einen Kontrollprozessor und zwei Arbeitsprozessoren

In der Anlaufphase wird auf eine hohe Zahl gespeicherter Datenblöcke pro Arbeitsprozessor Wert gelegt, da dann - bedingt durch die Fülle der gespeicherten Zwischenergebnisse - erwartet werden kann, daß der Kontrollprozessor in Kernphase und Auslaufphase nur selten auf einen Arbeitsprozessor warten muß.

Wenn die zu verarbeitenden Datenblöcke fast gleich lang sind und die Anteile für Ein- und Ausgabe von Datenblöcken gegenüber dem Anteil für die restliche Verarbeitung überwiegen, kann bei Verwendung der Round-Robin-Heuristik unter Umständen bereits mit einem oder zwei gespeicherten Datenblöcken pro Arbeitsprozessor ein Warten auf Ergebnisse durch den Kontrollprozessor vermieden werden. Falls nur ein Datenblock pro Arbeitsprozessor gespeichert werden muß, werden Verzögerungen durch Konflikte beim Zugriff auf den gemeinsamen Speicher gänzlich vermieden. Falls sich Warten auf dem Kontrollprozessor durch Abspeichern von zwei Datenblöcken pro Arbeitsprozessor vermeiden läßt, können Speicherkonflikte durch Verwendung von Doppelpuffern ausgeschaltet werden.

3.2.3 Eine zustandsorientierte Heuristik

Die Round-Robin-Heuristik des Kap. 3.2.2 führt (vgl. Abb. 3.9) aufgrund variierender Zeiten zur Verarbeitung von Datenblöcken durch Arbeitsprozessoren manchmal bereits in der Kernphase zu beträchtlichen Wartezeiten. Insbesondere ergeben sich für eine kleine Menge gespeicherter Datenblöcke pro Arbeitsprozessor Zeitverluste durch Warten auf einen bestimmten Prozessor, die sinnvoller genutzt werden können.

Mögliche Wartephasen lassen sich verkürzen (unter Umständen sogar vermeiden), wenn in der Kernphase nicht immer auf einen bestimmten Prozessor gewartet werden muß, sondern lediglich auf den, der als nächster ein Ergebnis anliefert. Dieses muß nun nicht in jedem Fall zur Ausgabe verwertbar sein; doch läßt sich dessen Existenz zumindest vermerken, so daß zum Zeitpunkt, zu dem eine Ausgabe möglich ist, nicht mehr darauf gewartet werden muß. Zwischenzeitlich kann jedoch der fertig gewordene Arbeitsprozessor, der möglicherweise zum Stillstand kommt, mit einem neuen Datenblock versorgt werden.

Unter Berücksichtigung dieser Vorgehensweise entstand die zustandsorientierte Heuristik. Sie durchläuft wie die Round-Robin-Heuristik zunächst die Anlaufphase, wartet dann jedoch nicht auf das Ergebnis eines bestimmten Arbeitsprozessors, sondern auf das Ergebnis des Prozessors, welches überhaupt als nächstes vorliegt. Die Existenz dieses Ergebnisses wird vermerkt.

Anschließend werden so lange reihenfolgerichtig alle vermerkten Ergebnisse ausgegeben, bis ein auszugebendes Ergebnis als nicht vermerkt festgestellt wird. Dann wird dem Arbeitsprozessor ein neuer Datenblock zur Verarbeitung zugeordnet, der das zuletzt vermerkte Ergebnis anlieferte, wenn dies aufgrund der Menge an freiem Platz zur Aufnahme eines neuen Datenblocks möglich ist.

Abb. 3.10 zeigt anhand eines Flußdiagramms die Kernphase der zustandsorientierten Heuristik. Abb. 3.11 zeigt die Verarbeitung des Programmsystems der Abb. 3.3 mit der zustandsorientierten Heuristik durch einen Kontrollprozessor und zwei Arbeitsprozessoren.

Die Auslaufphase stimmt mit der der Round-Robin-Heuristik überein.

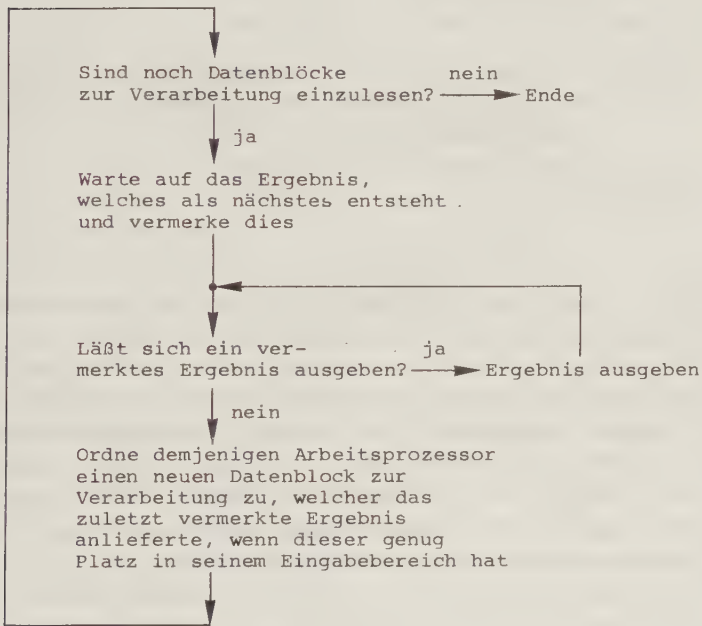


Abb. 3.10: Kernphase der zustandsorientierten Heuristik

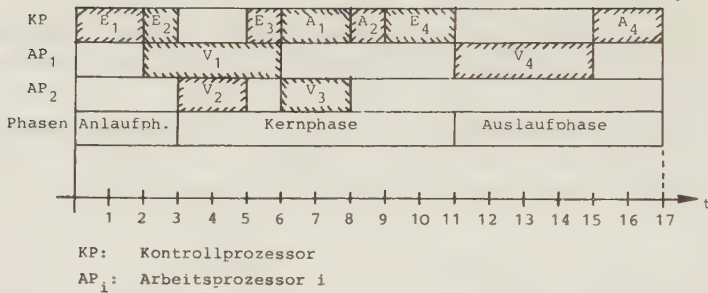


Abb. 3.11: Verarbeitung des Aufgabengraphen der Abb. 3.3 mit der zustandsorientierten Heuristik durch einen Kontrollprozessor und zwei Arbeitsprozessoren

3.2.4 Eine speicherorientierte Heuristik

Die beiden bislang betrachteten Heuristiken haben vor allem für eine kleine Zahl gespeicherter Datenblöcke den entscheidenden Nachteil, daß in der Kernphase Wartezeiten auf dem Kontrollprozessor entstehen, obwohl in dieser Zeit aufgrund der Arbeitsauslastung der Arbeitsprozessoren irgendeinem Arbeitsprozessor weitere Datenblöcke zur Verarbeitung zugeteilt werden könnten. Sie berücksichtigen auch nicht genügend die Tatsache, daß die zu verarbeitenden Datenblöcke stark in ihrer Größe variieren können oder verschiedene Arbeitsprozessoren unterschiedlich lang für die Bearbeitung des gleichen Datenblocks brauchen können.

Während im Fall unterschiedlich "schneller" Arbeitsprozessoren bei Verwendung der Round-Robin-Heuristik lediglich mit Verzögerungen durch den "langsamsten" Arbeitsprozessor gerechnet werden muß, können bei Verwendung der zustandsorientierten Heuristik zusätzlich dadurch Verzögerungen entstehen, daß Ein- und Ausgabepuffer des "schnellsten" Arbeitsprozessors überzulaufen drohen.

Dem Effekt, daß Parallelprogramme durch Prozessoren mit unterschiedlichen Geschwindigkeiten für die Bearbeitung gleicher Teilaufgaben verlangsamt werden, läßt sich mit Zuteilungsstrategien begegnen, die dem schnellsten Prozessor die Teilaufgaben auf dem kritischen Pfad des Aufgabengraphen (dem Pfad mit der längsten Bearbeitungszeit) zuordnen [Dem82].

Round-Robin-Heuristik und zustandsorientierte Heuristik gehen von der allerdings realistischen Annahme aus, daß für eine genügend große Zahl gespeicherter Datenblöcke sich Länge und damit Bearbeitungszeit zur Transformation so verteilen, daß ein in etwa gleicher Arbeitsanfall für alle Arbeitsprozessoren entsteht. Die beiden Heuristiken werten Informationen lediglich über die Zahl, nicht jedoch über den Umfang der gespeicherten Datenblöcke aus.

Die speicherorientierte Heuristik versucht, in der Kernphase den Eingabebereich der Arbeitsprozessoren immer auf einem möglichst hohen Stand zu halten. Dabei wird die Strategie verfolgt, bei Neuvergabe eines Datenblocks immer den Arbeitsprozessor zu berücksichtigen, der im Augenblick in seinem Eingabepuffer den meisten Platz hat.

Zur Vermeidung von Pufferüberläufen wird wie bei den Heuristiken der Kap. 3.2.2 und 3.2.3 ein Wert für die Belegung des Eingabebereichs eines Arbeitsprozessors festgelegt, ab dem dem betreffenden Arbeitsprozessor kein neuer Datenblock mehr zur Verarbeitung zugeteilt werden darf. Ein Arbeitsprozessor bekommt in jedem Fall einen neuen Datenblock zugewiesen, wenn der Kontrollprozessor dessen Eingabepuffer als "leer" erkennt.

Damit der Kontrollprozessor möglichst frühzeitig in der beschriebenen Form auf Veränderungen bezgl. der Menge der gespeicherten Datenblöcke auf den Arbeitsprozessoren reagieren kann, wird obige Strategie immer nach Ausgabe höchstens eines Ergebnisses anzuwenden versucht. Es wird auch nicht auf irgendeinen Prozessor gewartet, wenn beim Versuch der Ausgabe das angeforderte Ergebnis noch nicht bereitsteht, sodaß der Kontrollprozessor für diese Heuristik in der Kernphase ständig

aktiv ist und unter Umständen bei auftretenden Engpässen aktiv wartet.

Abb. 3.12 zeigt anhand eines Flußdiagramms die Struktur der Kernphase der speicherorientierten Heuristik. Abb. 3.13 zeigt die Verarbeitung des Programmsystems der Abb. 3.3 mit der speicherorientierten Heuristik durch einen Kontrollprozessor und zwei Arbeitsprozessoren.

Für Schleifendurchläufe in der Kernphase, in denen weder ein Ergebnis ausgegeben werden kann noch ein neuer Datenblock auf einen Arbeitsprozessor gelesen werden kann (also aktiv gewartet wird), empfiehlt es sich, den Kontrollprozessor in die Arbeitsphase mit einzubeziehen. Durch Lauftests läßt sich ermitteln, welche Menge an Eingabedaten der Kontrollprozessor verarbeiten kann, um eine möglichst geringe Gesamtlaufzeit mit möglichst wenig Wartezeiten auf allen Prozessoren zu erzielen (vgl. Beispiel in Kap. 4.2).

In Programmsystemen, bei denen keine Reihenfolge vorgeschrieben ist, in der Ergebnisse ausgegeben werden müssen, kann der Kontrollprozessor an Arbeitsprozessoren vergebene Datenblöcke von "hinten" in Nachbarschaftshilfe mit bearbeiten. Wenn ein Arbeitsprozessor die Teilaufgaben T_{i1} bis T_{ik} zu bearbeiten hat, kann der Kontrollprozessor diese in der Reihenfolge T_{i1}^k bis T_{i1}^1 bearbeiten, eine Verarbeitungsform, die in [Kung76] als adaptiv bezeichnet wird. Über Zustandsabfragen kann dafür gesorgt werden, daß dem Arbeitsprozessor geholfen wird, der am meisten in Rückstand liegt und daß keine Teilaufgabe doppelt ausgeführt wird.

Welche der beschriebenen Heuristiken verwendet wird, ist neben der Struktur des Programmsystems (insbesondere davon, über welche Bereiche die Ausführungszeiten der einzelnen Teilaufgaben wie stark streuen) auch von der verfügbaren Hardware (vgl. Kap. 3.3) abhängig und davon, welche Möglichkeiten der Kommunikation bzw. von Fremdspeicherzugriffen bestehen und wie "teuer" deren Gebrauch ist (vgl. Kap. 5 und 6). So sind z. B. die speicherori-

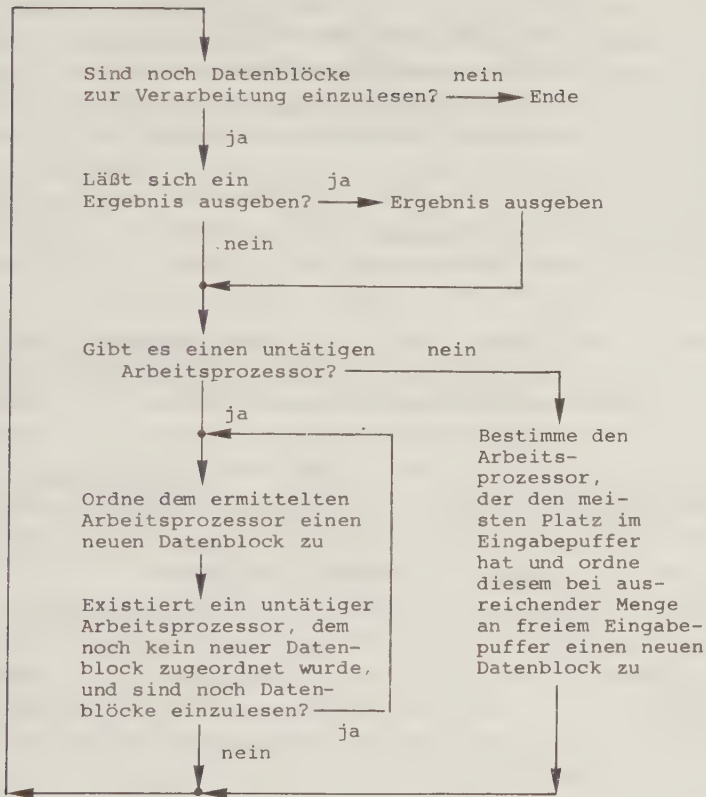


Abb. 3.12: Kernphase der speicherorientierten Heuristik

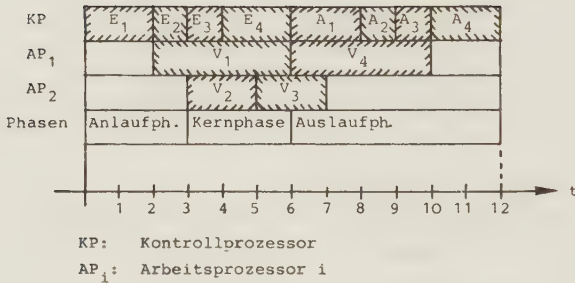


Abb. 3.13: Verarbeitung des Aufgabengraphen der Abb. 3.3 mit der speicherorientierten Heuristik durch einen Kontrollprozessor und zwei Arbeitsprozessoren

enterte Heuristik und adaptive Implementierungen aus Effizienzgründen sinnvoll nur mit Systemen ausführbar, bei denen zwischen miteinander kommunizierenden Prozessoren eine Kopplung über einen gemeinsam benutzbaren Speicher besteht.

[Röb79] beschreibt Methoden zur Analyse von Parallelalgorithmen, die durch statische Dekomposition (Strategien vom Typ der Round-Robin-Heuristik) und dynamische Dekomposition (Heuristiken in Abhängigkeit von Zuständen innerhalb des Programmlaufs) entstehen. In [LiAg81] wird eine Zusammenstellung von Modellen als Grundlage zum Entwurf von Heuristiken zur Ablaufsteuerung von Parallelprogrammen gegeben.

3.3 Zur Frage einer geeigneten Rechnerarchitektur

In den Kapiteln 3.1 und 3.2 wurde bei der Konstruktion paralleler Verarbeitungsstrategien für ordnungserhaltende Programmsysteme von einer hierarchischen EGPA-ähnlichen Rechnerarchitektur ausgegangen. Bei der Zuordnung von Programmfunktionen zu einzelnen Prozessoren wurden von dieser Struktur jedoch nur zwei Verbindungen vom Kontrollprozessor zu den Arbeitsprozessoren und einige Verbindungen der Arbeitsprozessoren untereinander benötigt. Bei Verfahren auf der Grundlage der Aufteilung des zu transformierenden Datenstroms auf die Arbeitsprozessoren wurden die Verbindungen der Arbeitsprozessoren untereinander überhaupt nicht ausgenutzt.

Ein Nachteil der betrachteten Struktur besteht darin, daß nur ein Prozessor (der Kontrollprozessor) die Fähigkeit besitzt, Daten einzulesen und Daten auszugeben. Dies hatte zur Folge, daß parallele Verarbeitungsstrategien nicht nur das Problem der Zuordnung von Aufgaben zu Prozessoren zu berücksichtigen hatten, sondern auch das der Bestimmung der Reihenfolge, in der Ein- und Ausgaben ausgeführt werden sollten.

Diese Feststellungen werfen die Fragen nach einer für diese Aufgabenstellung besser geeigneten Rechnerarchitektur auf. Eine Optimierung in dieser Richtung hat die Mindestzahl von Prozessoren, Speichern und Verbindungen zu finden, um die gestellte Aufgabe in Minimalzeit auszuführen. Die folgenden Vorschläge zeigen, wie durch "bessere" Rechnerarchitekturen die Laufzeit der vorgestellten Parallelverfahren verringert werden kann und welche Vereinfachungen sich dadurch bezgl. der zu implementierenden Strategien ergeben.

Der erste Vorschlag besteht darin, eine zweite Schicht von Prozessoren an die Arbeitsprozessoren zu hängen und auf diese die einzelnen Funktionen der Teilaufgabe "Verarbeitung" aufzuteilen. Dies bringt bei pipelineartiger Verarbeitung auf der Ebene der Arbeitsprozessoren den in Kap. 3.1 beschriebenen Geschwindigkeitsgewinn. Diese Form wird jedoch nur dann einen Geschwindig-

keitsgewinn für die Ausführung der Gesamtaufgabe bringen, wenn der Teil "Verarbeitung" gegenüber den Teilen "Eingabe" und "Ausgabe" überwiegt. Eine Vereinfachung von Parallelisierungsstrategien ergibt sich nicht.

Eine zweite Möglichkeit, durch Änderung der Rechnerarchitektur für ordnungserhaltende Programmsysteme einen Geschwindigkeitsgewinn zu erzielen, besteht darin, der hierarchischen Struktur einen Kontrollprozessor zur Ausgabe hinzuzufügen. Zusätzliche Speicher zur Aufnahme der von den Arbeitsprozessoren erzielten Ergebnisse, auf die der Ausgabeprozessor Zugriff hat, vermindern mögliche Verzögerungen durch Konflikte beim Zugriff auf gemeinsam benutzte Speicher. Abb. 3.14 zeigt die Struktur.

Das Programm für den Eingabeprozessor vereinfacht sich dadurch, daß jetzt keine Berücksichtigung von Ausgaben in der Bearbeitung erforderlich ist. Es findet eine Reduzierung auf das Problem statt, welchem Arbeitsprozessor der nächste zu transformierende Datenblock zugeordnet werden soll. Dazu kann eine der in Kap. 3.2 beschriebenen Heuristiken verwendet werden.

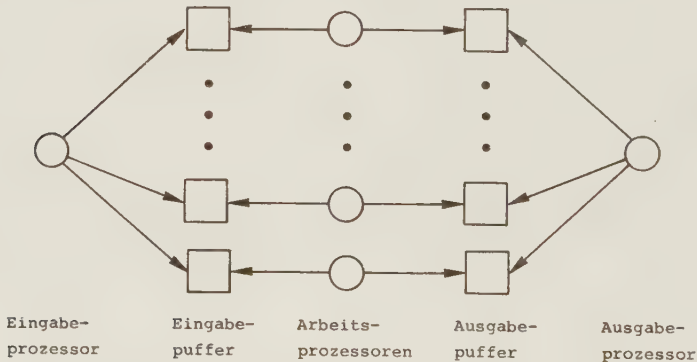


Abb. 3.14: Ausgabeprozessor und zusätzliche Speicherbereiche zur Verringerung der Zeit zur Ausführung ordnungserhaltender Programmsysteme

Der Ausgabeprozessor führt nur noch die Auslaufphase durch. Wenn eine statische Zuordnung von zu transformierenden Datenblöcken zu Arbeitsprozessoren erfolgt, braucht der Ausgabeprozessor keine weiteren Informationen zur Erkennung der richtigen Ausgabereihenfolge mehr. Im Fall einer dynamischen in Abhängigkeit von Programmzuständen erfolgenden Zuordnung von zu transformierenden Datenblöcken zu den Arbeitsprozessoren kann vom Eingabeprozessor im Partitionierungsprogramm Information über die Nummer des Datenblocks mitgegeben werden, die durch die Arbeitsprozessoren an den Ausgabeprozessor weitergereicht wird. In letzterem Fall wird das Programm auf dem Ausgabeprozessor dem "Ausgabeteil" in der Kernphase der zustandsorientierten Heuristik entsprechen.

Eine weitere Vereinfachung der Parallelstrategien für die Ausführung ordnungserhaltender Programmsysteme ergibt sich, wenn der Eingabeprozessor die zu transformierenden Datenblöcke auf einem Speicher ablegt, der von allen übrigen Prozessoren erreichbar ist (vgl. Abb. 3.15).

In diesem Fall ist eine dezentral organisierte Verarbeitung möglich. Wenn ein Arbeitsprozessor die Transformation eines Datenblocks durchgeführt hat, nimmt er sich selbsttätig vom gemeinsamen Speicher den nächsten zu verarbeitenden Datenblock.³¹⁾ Synchronisationsroutinen sind erforderlich, um zu verhindern, daß ein Datenblock mehrfach transformiert wird.

Diese Organisationsform erlaubt auch, daß Eingabe- und Ausgabeprozessoren bei drohender Untätigkeit bei der Verarbeitung mithelfen. In [GoRa72] wird über Simulation gezeigt, daß dezentrale Organisation bei der parallelen Ausführung von Teilaufgaben einer zentralen Organisation im allgemeinen überlegen ist, da der durch Kontrolloperationen bedingte Overhead geringer wird und über verschiedene Prozessoren verteilt werden kann.

31) In [Gilo82] wird diese Verarbeitungsform als "Auftragsanziehung" bezeichnet. Verarbeitungsstrategien des Kap. 3.2 werden dort in die Klasse der Verfahren der "Auftragszuweisung" eingeordnet.

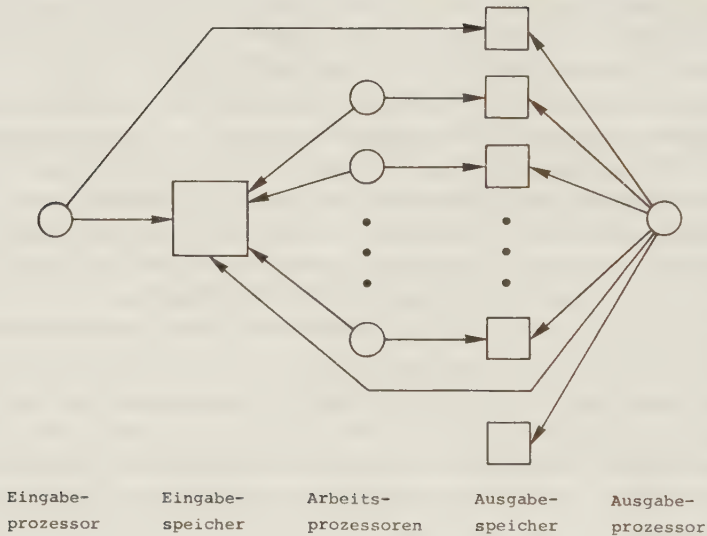


Abb. 3.15: Architektur mit einem gemeinsamen Speicher zur Vereinfachung von Strategien für die Ausführung ordnungserhaltender Programmsysteme

Der gemeinsame Speicher kann natürlich über Interferenzen beim Zugriff von verschiedenen Prozessoren aus zu einem Engpaß werden. Die Vorgehensweise bei der Implementierung (autonome Ausführung einer "großen" Teilaufgabe auf einem Prozessor mit "wenigen" Zugriffen zu Fremddaten) läßt diese Gefahr jedoch gering erscheinen. Entsprechende Vermutungen werden durch die in [From82] durchgeführte Modellierung der Speicherinterferenz und die in [Horn84] beschriebenen Messungen des Verhältnisses zwischen Zugriffen auf den eigenen Speicher und auf einen Fremdspeicher selbst in "E/A-intensiven" Phasen bestätigt.

Die beste Struktur aus Prozessoren und Speichern zur Ausführung von nach Verfahren der Kap. 2.2 und 2.3 konstruierten ordnungserhaltenden Programmsystemen ergibt sich natürlich dann, wenn genügend Prozessoren, Speicher und Verbindungselemente vorhanden sind, sodaß jeder Teilaufgabe im Verarbeitungsteil des Graphen der Abb. 2.15 ein eigener Prozessor zugeordnet werden kann. (Für Eingaben und Ausgaben reicht aufgrund der Abhängigkeitsstruktur jeweils ein Prozessor aus.) In diesem Fall sind auch keine Überlegungen zur Optimierung von parallelen Verarbeitungsstrategien notwendig. Die Ausführung der Anlaufphase der Round-Robin-Heuristik auf dem Eingabeprozessor und der Auslaufphase auf dem Ausgabeprozessor führt bereits zur optimalen Lösung in bezug auf die Aufgabenstruktur.

Diese Architektur wird dann zwar bezgl. der kürzestmöglichen Verarbeitungszeit immer die optimale Strategie ermöglichen, bezgl. der Effizienz des Gesamtsystems, definiert als Quotient zwischen dem gegenüber der Einzelprozessorlösung erreichten Geschwindigkeitsgewinn und der Zahl der im System vorhandenen Prozessoren, aber sicher keinen guten Wert liefern.

Man muß sich allerdings bei der Betrachtung des Verhältnisses zwischen Hardware- und Softwarekosten bei der Realisierung großer Systeme (die sich immer mehr zu Lasten der Software verschieben) fragen, ob angesichts einer Vereinfachung der Software bei ausreichend großen Systemen die Frage der Effizienz noch die dominierende Rolle spielen sollte.

Die Frage nach der besten Architektur zur Ausführung eines Parallelprogramms läßt sich generell nur in Abhängigkeit der Struktur der Anwendung beantworten. Ein Rechnerbaukasten, der ein Zusammenstecken eigener Verbindungen zwischen Prozessoren und Speichern gestattet [HäRo80], kann in Ergänzung zu der in Kap. 2 entwickelten Methodik zur Erstellung von Parallelprogrammen sicher ein wertvolles praktisches Hilfsmittel zum Auffinden einer für eine spezielle Aufgabe mit einer gegebenen Zerlegung in Teilaufgaben optimal geeigneten Rechnerarchitektur sein.

4 Anwendungsbeispiele

Die nachfolgende Beschreibung zweier auf dem EGPA-System durchgeführter Implementierungen gibt in Form einer Fallstudie eine Bewertung der in den Kapiteln 2 und 3 eingeführten Methode zur Parallelisierung und den darauf aufbauenden Verarbeitungsstrategien.

Das in Kap. 4.1 dokumentierte Beispiel "Textformatierung" steht dabei für ein "großes" ordnungserhaltendes Programmsystem. An diesem wird neben einer Bewertung der in Kap. 3 dargestellten parallelen Verarbeitungsstrategien insbesondere verdeutlicht, welche Rolle Strukturen und Verarbeitungszeiten von Teilaufgaben bei der Bestimmung einer informationskonsistenten Folge von Datenblöcken und deren Entflechtung spielen.

Das in Kap. 4.2 dokumentierte Beispiel "Netzwerkanalyse" steht für ein "kleines" ordnungsfreies Programmsystem. Es zeigt, wie beschriebene Methoden innerhalb ordnungsfreier Programmsysteme einsetzbar sind und welche weiteren Möglichkeiten speziell für diese Systeme bestehen. Schließlich gibt dieses Beispiel auch einen Vergleich der in Kap. 2 dokumentierten Methode mit einer "direkten" Methode zur Implementierung von Parallelprogrammen und kann damit Geltungsbereich und Grenzen des Ansatzes verdeutlichen.

4.1 Textformatierung

Das an der EGPA-Anlage implementierte Textverarbeitungssystem [KnRa82] hat sich neben Gesichtspunkten struktureller Art vor allem deshalb zu Untersuchungen bezgl. einer Parallelisierung angeboten, weil es (neben dem Editor) eines der meistverwendeten Benutzerprogramme (also kein "synthetisches" Programm) im EGPA-System ist. Es diente nicht nur als Hilfsmittel bei der Erstellung von Veröffentlichungen und Examensarbeiten, sondern auch für die gesamte (ca. 800 Seiten umfassende) EGPA-Dokumentation.

Obwohl sich aus Überlegungen bezgl. einer Verkürzung der Ausführungszeit durch Einsatz eines Multiprozessors dieses Beispiel kaum eignet, - die Grenzen eines möglichen Geschwindigkeitsgewinns werden in einem hierarchischen Multiprozessorsystem bereits bei Verwendung von vier Arbeitsprozessoren erreicht -, zeigt sich doch gerade hier, welche Bedeutung der in den Kap. 2 und 3 dokumentierte systematische Ansatz besitzt.

Das sequentielle Programmsystem umfaßt ca. 5000 Programmzeilen und hat allein aufgrund seines Umfangs schon Methoden der Parallelisierung auf der Grundlage einzelner Anweisungen ausgeschlossen.

Nachfolgend wird von der Programmstruktur ausgehend die Anwendung der in den Kap. 2 und 3 dokumentierten Verfahrensschritte gezeigt und anschließend die Ergebnisse der implementierten Parallelisierungsstrategien kommentierend bewertet.

4.1.1 Programmstruktur und Alternativen zur Parallelisierung

Im Gegensatz zu vielen heute in Rechensystemen implementierten Textverarbeitungssystemen (eine Übersicht über Konzepte und Realisierungen findet sich in [FuSS82]) ist im zugrundeliegenden System eine vollständige Trennung der Teile "Texteingabe", "Textformatierung" und "Textausgabe" vollzogen worden. Eingabetexte bestehen aus Textteilen, in die formatfrei Formatierkommandos wie ";einruecke", ";bild", ";fussnote" und dergleichen eingestreut sind. Praktisch bedeutet dies, daß eine Textformatierung erst dann begonnen werden kann, wenn die Texteingabe abgeschlossen worden ist und daß eine Ausgabe erst nach vollständiger Formatierung erfolgt.

Dies ist für die Erreichung eines Geschwindigkeitsgewinns durch Bearbeitung auf einem Multiprozessorsystem wesentlich, da Ein- und Ausgabe nicht von der Geschwindigkeit des Rechensystems abhängen. Der Teil "Textformatierung" hat die in Abb. 4.1 gezeigte Gestalt.

Die unmittelbar aus Abb. 4.1 ersichtliche Parallelisierung (parallele Ausführung der drei Funktionen ZEILE für Text, Bilder und Fußnoten) bringt in der Regel deshalb keinen Geschwindigkeitsgewinn, da Text, Bilder und Fußnoten niemals die gleiche Größe haben. Im Normalfall überwiegt der Teil "Text" gegenüber den Anteilen "Abbildungen" und "Fußnoten".

Nach der gemäß Kap. 2.2 gestalteten Form der Module SYMBOL, SYMBOL2, VORLAUF, ZEILE und SEITE als E/A-orientierte Transformationsmodule mit Kontrolle wurde bei der Zuordnung von ganzen Programmfunktionen zu den Prozessoren und Speichern des EGPA-Systems die in Abb. 4.2 dargestellte Struktur untersucht.

Dafür waren folgende Gründe maßgebend:

1. Auf keinem Prozessor sollte mehr als ein Prozeß laufen, da ein Multiprogramming auf der Ebene der A-Prozessoren nicht vorgesehen ist.
2. Ein- und Ausgaben werden vom B-Prozessor durchgeführt.³²⁾ Da

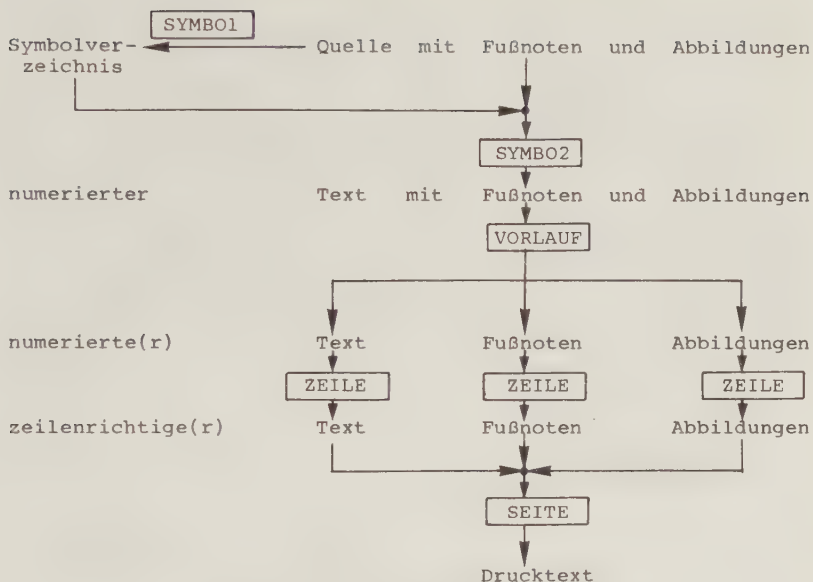


Abb. 4.1: Struktur des Programmsystems zur Textformatierung

SYMBOL2 als eine Eingabe die ursprüngliche Datei mit dem editierten Text von der Peripherie einlesen muß, wurde aus Effizienzgründen die Funktion SYMBOL2 dem B-Prozessor zugeordnet.

-
- 32) Im EGPA-Pilotsystem ist es zwar möglich, auch innerhalb der Programme auf den A-Prozessoren E/A-Aufträge zu formulieren, das EGPA-Betriebssystem beauftragt aber den B-Prozessor mit der Durchführung des E/A-Vorgangs, da nur dieser Zugriff zur Peripherie hat.

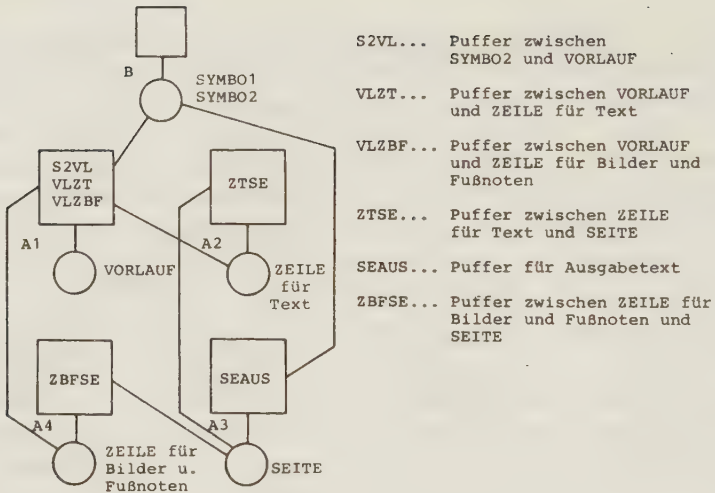


Abb. 4.2: Aufteilung der Funktionen und Ergebnisse der Textformatierung auf Prozessoren und Speicher der EGPA-Pilotpyramide

3. Da das gesamte von SYMBOL1 erzeugte Symbolverzeichnis als Eingabe für SYMBOL2 benötigt wird, kann SYMBOL1 nicht parallel zu SYMBOL2 laufen. Deshalb kann SYMBOL1, ohne daß Geschwindigkeitseinbußen zu befürchten sind, vom B-Prozessor ausgeführt werden.
4. Die Funktionen ZEILE zur Bearbeitung von Abbildungen und Fußnoten können deshalb in einem Prozeß zusammengefaßt werden, da die dort transformierten Abbildungen und Fußnoten von der nachfolgenden Funktion SEITE in der gleichen Reihenfolge angefordert werden, in der sie von der vorangehenden Funktion VORLAUF ausgeschrieben werden.

5. Der Speicher des die Funktion VORLAUF ausführenden Prozessors A1 nimmt deshalb die meisten Zwischenergebnisse auf, da der Code der VORLAUF-Funktion im Vergleich zu dem der Funktionen ZEILE und SEITE den geringsten Platz einnimmt.

Für Texte ohne Abbildungen und Fußnoten ergibt sich damit eine reine Pipeline für die Textformatierung, sonst ergibt sich eine pipelineartige Verarbeitung, die für einen Lauf zwei Verzweigungen enthält.

Zur Bestimmung der mit dieser Konfiguration erreichbaren Geschwindigkeitssteigerung gemäß der in Kap. 3.1 abgeleiteten Formel werden die Bearbeitungszeiten der einzelnen Teilaufgaben benötigt, die in Abb. 4.3 für das Beispiel der Formatierung eines 47 Seiten langen Textes aufgeführt sind.

Modul	Laufzeit in Sekunden	Laufzeit in Prozent
SYMBOL1	15.6	4.7
SYMBOL2	42.9	13.0
VORLAUF	49.8	15.0
ZEILE	163.4	49.4
SEITE	56.4	17.1
andere	2.6	0.8

Abb. 4.3: Laufzeit der Funktionen für die Textformatierung am Beispiel der Aufbereitung eines Textes im Umfang von 47 Seiten

Die Zeiten zeigen, daß mit dieser Form der Parallelisierung (bedingt durch den Anteil der Funktion ZEILE an der Gesamtlaufzeit) im vorliegenden Beispiel bestenfalls ein Geschwindigkeitsgewinn um den Faktor 2 möglich ist. Wenn reine CPU-Zeiten (d. h. ohne Einflüsse durch E/A) berücksichtigt werden, liegt der erreichbare Geschwindigkeitsgewinn noch geringer (etwa bei 1.5), da dann der Modul ZEILE ca. 67 % der Gesamtzeit verbraucht.

Ein fairer Vergleich ist jedoch schon deshalb nicht möglich, da zur pipelineartigen Verarbeitung keine entsprechende sequentielle Verarbeitungsform existiert.³³⁾ Bei einer Aufteilung der zu verarbeitenden Datenmenge über verschiedene Prozessoren ist eine vollständige Textformatierung dann möglich, wenn der Text an Seitengrenzen geteilt wird. (Bei keiner anderen Partitionierung des Eingabetextes ist eine Parallelarbeit über verschiedene Textteile durch den Seitenumbruch möglich.) Dies setzt jedoch voraus, daß a priori bekannt ist, welche Textteile an einem Seitenanfang zu stehen kommen oder einigermaßen oft (über ein entsprechendes Kommando) ein Seitenwechsel erzwungen wird. Da keine der beiden Annahmen realistisch ist, muß eine per Programm vorzunehmende Partitionierung des Textes in Datenblöcke erfolgen, die vorher unbekannt ist. Dies bedingt, daß ein Seitenumbruch nicht über Textteile, sondern nur über den gesamten Text (bzw. die reihenfolgerichtige Zusammenfassung aller Textteile) möglich ist.

Die Programme SYMBO2 bis ZEILE können jedoch auf verschiedenen Textteilen parallel (nebenläufig) ablaufen, wenn sichergestellt ist, daß die von diesen Programmen erzeugten Stücke wieder vollständig zum richtigen Ausgabertext zusammengesetzt werden können. Dies ist möglich, wenn der Eingabetext in Absätze aufgeteilt wird. Absätze sind also Datenblöcke, die bezgl. der Textformatierung zu einer informationskonsistenten Folge zusammengesetzt werden können.

33) Eine zu einer pipelineartigen Verarbeitung entsprechende sequentielle Verarbeitungsform wäre, Funktionen vollständig auszuführen und die jeweiligen Ergebnisse im Hauptspeicher abzulegen. Dies ist jedoch aus Gründen der Kapazität des Hauptspeichers im allgemeinen nicht möglich. Andere Formen (vgl. Kap. 5.4.2) setzen voraus, daß sequentielle Programme mit mehreren Prozessen formuliert werden.

Die folgende Zusammenstellung zeigt, was das den Eingabetext in Absätze partitionierende Programm zur Entflechtung verschiedener Absätze zu berücksichtigen hatte.

1. Nach Durchlaufen der Initialisierung der Routine VORLAUF wird die nächste einzugebende Zeile als zum Text zugehörig interpretiert, d. h. der Eingabetext darf nicht innerhalb von Abbildungen oder Fußnoten geteilt werden.
2. Nach Durchlaufen der Initialisierung der Routine ZEILE wird davon ausgegangen, daß der folgende Text nicht eingerückt ist und im Modus "Verarbeitung mit Randausgleich" weiter verarbeitet wird. Das in die Eingabe integrierte Partitionierungsprogramm wurde dazu so gestaltet, daß ein von der Norm abweichender Zustand zum Ende eines Absatzes ("Text eingerückt" oder Modus "Verarbeitung ohne Randausgleich") in Form von Kommandos an den nächsten Absatz weitergegeben wird. Ein entsprechendes Verfahren wurde auch für aufeinanderfolgende Bilder/Fußnoten durchgeführt.
3. Die Funktion SEITE ist so gestaltet, daß fünf Textzeilen im voraus gelesen (und gepuffert) werden, bevor eine bearbeitete Zeile ausgegeben wird. Dies gilt jedoch nicht für Abbildungen und Fußnoten. Damit im aktuellen Absatz enthaltene Abbildungen und Fußnoten mit verarbeitet werden, müssen nach Ende der entsprechenden Einheit noch mindestens fünf Textzeilen folgen. Das Partitionierungsprogramm mußte also neben einer Endeerkennung für derartige Einheiten eine Abschätzung der Zahl folgender Zeilen bis zum nächsten Absatzende durchführen.

Diese Zusammenstellung zeigt, daß eine Entflechtung unter Umständen sehr viel mehr Aktionen umfaßt, als lediglich Datenbestände zu partitionieren. Für das Beispiel der Textformatierung ist im Gegensatz zur Gestaltung paralleler Compiler auf der Basis der Aufteilung des Datenbestandes [Mes82] keine Partitionierung an beliebigen Stellen der Eingabedatenmenge möglich, da die die Datenblöcke interpretierenden Programme zur Erzeugung ihrer Ausgaben keine Informationslücken haben dürfen.³⁴⁾

Es stellt sich bei der notwendigen Einbringung eines die Eingabedaten analysierenden Vorinterpretierers die Frage, ob ein solches Programm nicht selbst zu viel Zeit braucht, sodaß dadurch der Parallelisierungserfolg in Frage gestellt ist.

In diesem Beispiel liegt die Zeit zum Einlesen des Textes einschließlich dessen Partitionierung in Absätze in der Größenordnung der Verarbeitungszeit, die für den Seitenumbruch benötigt wird, also erheblich unter der Verarbeitungszeit der parallel ausführbaren Teilaufgaben, sodaß in diesem Fall der Einbau des Vorinterpretierers tragbar ist.

Abb. 4.4 gibt eine Zusammenstellung der Verarbeitungszeiten, die

Teilaufgabe	Laufzeit in Sekunden	Laufzeit in Prozent
EINGABE	22.1	11.3
VERARBEITUNG	140.4	71.5
AUSGABE	21.8	11.1
Org. Arbeiten	12.0	6.1
Gesamt	196.3	100.0

Abb. 4.4: Laufzeit der Teilaufgaben für die Textformatierung auf der Basis von Absätzen (Beispiel: Aufbereitung eines Textes im Umfang von 47 Seiten)

bei der Textformatierung auf der Basis von Absätzen für die verschiedenen Teilaufgaben auftreten. Die Zeiten, die mit dem in [Klei82] bzw. [Müh84] beschriebenen integrierten Software-Meßsystem ISMS erhalten wurden, lassen die folgenden Schlußfolgerungen zu:

1. Die Anteile für Eingabe (Einlesen des Textes und Partitionie-

- 34) Entsprechendes gilt für alle Programmsysteme, die ihren Bestand an Eingaben in einer syntaktischen Form statt wie bei Compilern üblich in einer semantischen Form interpretieren.

rung in Absätze), Ausgabe (Seitenumbruch und Ausschreiben des formatierten Textes) und organisatorische Arbeiten (Initialisierung und Abschluß) benötigen nur 28.5 % der Gesamtzeit für die Formatierung. Wenn es gelingt, den Anteil für Verarbeitung (Ausführung von SYMBO2, VORLAUF und ZEILE) so auf die Arbeitsprozessoren zu verteilen, daß der Kontrollprozessor durch diesen Teil nicht mehr verzögert wird, ist bestenfalls eine Geschwindigkeitssteigerung um den Faktor 3.5 ($100 / 28.5$) möglich. (Der tatsächliche Wert dürfte aber niedriger liegen, da im Multiprozessorsystem mit einem höheren Anteil für organisatorische Arbeiten gerechnet werden muß.)

2. Der Zeitanteil für die Verarbeitung ist 3.2 mal so hoch wie die Summe der Teile für Eingabe und Ausgabe. Somit dürfte in einem EGPA-ähnlichen System bei Verwendung von vier Arbeitsprozessoren für die Verarbeitung bereits das Maximum eines möglichen Geschwindigkeitsgewinns erreichbar sein. Dabei ist es nicht erforderlich, den Kontrollprozessor in den Verarbeitungsteil einzubeziehen.

In Kap. 4.1.2 werden die erzielten Ergebnisse in Abhängigkeit der verwendeten parallelen Verarbeitungsstrategie kommentierend bewertet.

Die hier am Beispiel des sequentiellen Programms vorgenommene Analyse zeigt nicht nur, daß auf der Basis der Aufteilung des Datenbestandes im allgemeinen höhere Werte für den Geschwindigkeitsgewinn durch Parallelarbeit zu erzielen sind. Es wird hier auch deutlich, daß sich bereits auf der Seite des sequentiellen Programms ein erheblicher Geschwindigkeitsgewinn dadurch erzielen läßt, daß Operationen für Ein- und Ausgabe auf Dateien eingespart werden (und dies, obwohl in diesem Fall die Programmverweilzeit durch Hinzunahme des Partitionierungsprogramms verlängert wird).

4.1.2 Ergebnisse von Parallelisierungsstrategien

Für die Darstellung der Ergebnisse der durchgeführten Implementierungen werden für die einzelnen Parallelisierungsverfahren jeweils die "besten" aus Meßreihen entstandenen Ergebnisse verwendet. Als Beispiel liegt die Aufbereitung eines Textes im Umfang von 47 Seiten zugrunde, der auch Grundlage für die Laufzeiten der sequentiellen Programme war. Neben einer Darstellung der erzielten Zeiten (bzw. Geschwindigkeitsgewinnen) wird der Einfluß bewertet, der im Parallelprogramm durch organisatorische Maßnahmen (vor allem Synchronisation) entsteht und in die Kommentierung einbezogen.

Bei der Verarbeitungsform "Zuordnung von Programmfunktionen zu Prozessoren" lag die Verarbeitungszeit bei etwa 126 sec und war unabhängig von der Größe der Puffer für Zwischenergebnisse und unabhängig davon, ob bei der Verarbeitung der Kontrollprozessor eine eingabeorientierte oder ausgabeorientierte Strategie³⁵⁾ verfolgte.

Die erhaltenen Resultate verdeutlichen, daß bei Zuordnung von Programmfunktionen zu Prozessoren praktisch keine Möglichkeiten bestehen, Lauf- und Wartezeiten durch "algorithmisch" spezialisierte Parametersteuerung zu minimieren.

Wesentliche Unterschiede bezgl. der Laufzeiten ergeben sich in diesem Fall jedoch in Abhängigkeit des verwendeten Typs der Puffer zur Aufnahme von Zwischenergebnissen und des verwendeten Mechanismus zur Synchronisation. In Kap. 5.2 werden anhand dieses Beispiels diese Einflüsse zusammengestellt und bewertet.

35) Bei der eingabeorientierten Strategie wurden Ergebnisse erst ausgegeben, wenn der Kontrollprozessor aufgrund von Pufferengpässen keine Eingabe mehr an Arbeitsprozessoren weitergeben konnte. Bei der ausgabeorientierten Strategie wurden durch den Kontrollprozessor erst dann neue Eingaben an Arbeitsprozessoren gegeben, wenn kein erzielttes Ergebnis mehr ausgegeben werden konnte, d. h. der Ausgabepuffer leer war.

Ganz im Gegensatz dazu gibt es eine Fülle von Parametern, die die Güte paralleler Verarbeitungsstrategien auf der Basis der Aufteilung der zu verarbeitenden Datenmenge bestimmen. In Kap. 3.2.1 wurde die Bedeutung fast gleicher Verarbeitungszeiten von Teilaufgaben zur Annäherung an optimale Lösungen nachgewiesen.

Im konkreten Fall (Textformatierung auf der Basis von Absätzen) besteht die Möglichkeit, dies durch Zusammenfassen mehrerer Absätze zu einem Hyper-Absatz zu erreichen. Hyper-Absätze sind in der Weise implementiert, daß eine bestimmte Mindestlänge an Zeilen vorgeschrieben wird. Abb. 4.5 gibt eine Übersicht über charakteristische Größen für die Textformatierung im sequentiellen Programm in Abhängigkeit der Mindestlänge von Hyper-Absätzen.

Abb. 4.5 zeigt, wie stark bereits im sequentiellen Programm für dieses Beispiel die Größe von zu verarbeitenden Datenblöcken

Mindestlänge eines Absatzes	1	5	10
Zahl aufbereiteter Absätze	1001	232	124
Größe des längsten Absatzes	33	36	46
Zeit zur Aufbereitung des kürzesten Absatzes in sec	0.06	0.22	0.13
Zeit zur Aufbereitung des längsten Absatzes in sec	3.78	3.99	6.56
Varianz in den Zeiten zur Aufbereitung von Absätzen	3.37	0.82	0.48
Verweilzeit im seq. Programm zur Formatierung in sec	233.17	200.40	196.30

Abb. 4.5: Charakteristische Größen für die Formatierung von Absätzen für verschiedene Mindestlängen von Hyper-Absätzen (Beispiel: Formatierung eines 47 Seiten langen Textes)

Verarbeitungszeiten beeinflusst. Sicher läßt sich sagen, daß mit wachsender Größe von Datenblöcken die Gesamtverarbeitungszeit abnimmt und daß im Fall, daß nur ein Datenblock zu transformieren ist, diese Zeit minimal wird. Doch läßt sich eine solche Form im allgemeinen aus Platzmangel nicht in der Weise realisieren, daß Zwischenergebnisse von Teilaufgaben auf dem Hauptspeicher gehalten werden können. Werden dazu Sekundärspeicher verwendet, fallen wiederum signifikante Zeiten für E/A-Vorgänge an (vgl. Abb. 4.3). Dies verdeutlicht, wie schwierig es für diese Art von Programmsystemen ist, die sequentielle Lösung zu finden, die als schnellste eine faire Vergleichsgrundlage für parallele Lösungen darstellt.

Für die folgenden Geschwindigkeitsvergleiche zwischen sequentiellen und parallelen Lösungen wurde im sequentiellen Programm und in den verschiedenen parallelen Fassungen immer von einer Größe eines Hyper-Absatzes ausgegangen, die mindestens 10 Zeilen umfaßt. Dies hat seinen Grund nicht nur in den oben aufgeführten Zeiten, sondern ist auch durch die Aufgaben- und Datenstruktur bedingt. So haben die überwiegende Zahl von Bildern und Fußnoten innerhalb eines zu formatierenden Textes etwa fünf Zeilen Umfang, die zusammen mit den fünf danach zu lesenden Zeilen zum Auffinden des Absatzendes (s. oben) die als zunächst willkürlich erscheinende Zahl von 10 für die Mindestgröße eines Hyper-Absatzes plausibel erscheinen lassen.

Zur Beurteilung, welche der in Kap. 3.2 beschriebenen Heuristiken für die Parallel-Implementierung die geeignete ist, sei Abb. 4.6 betrachtet, die zeigt, über welchen Bereich die Verarbeitungszeiten für die einzelnen Teilaufgaben bei der Formatierung von Absätzen streuen.

Die Zeiten legen den Schluß nahe, daß durch Spezialfälle der Round-Robin-Heuristik (Pufferung von einem oder zwei Datenblöcken pro Prozessor) kaum erreicht werden kann, daß der Kontrollprozessor zur Ausgabe eines Absatzes nicht zu warten braucht.³⁶⁾ Deshalb läßt sich in diesem Fall vermuten, daß parallele Strategien "besser" sind, die von einem hohen Bestand an gespei-

Teilaufgabe	Minimalzeit	Maximalzeit
Eingabe	0.02	0.53
Verarbeitung	0.06	5.47
Ausgabe	0.05	0.55

Abb. 4.6: Bereich der Verarbeitungszeiten (in sec) für die verschiedenen Teilaufgaben bei der Formatierung von Hyper-Absätzen mit einer Mindestlänge von 10 Zeilen

cherten Hyper-Absätzen ausgehen. Dies wird durch die in Abb. 4.7 dargestellten Ergebnisse der verschiedenen Heuristiken bestätigt. Die Ergebnisse, wie sie in der Graphik zum Ausdruck kommen, bestätigen die in Kap. 3.2 angestellten Vermutungen über den erreichbaren Geschwindigkeitsgewinn von wenig unter 3.5 und den Verlauf des Geschwindigkeitsgewinns in Abhängigkeit der Zahl gepufferter Datenblöcke pro A-Prozessor für die implementierten Heuristiken.

Für die Round-Robin-Heuristik ergibt sich bis zu einer Zahl von sechs gepufferten Datenblöcken pro Arbeitsprozessor in der Anlaufphase eine kontinuierliche Zunahme des Geschwindigkeitsgewinns, dann ein im wesentlichen konstanter Kurvenverlauf. Bei zwölf gepufferten Datenblöcken pro Arbeitsprozessor ergibt sich für alle Heuristiken eine Verschlechterung des Geschwindigkeitsgewinns, was auf auftretende Pufferengpässe zurückzuführen ist. Eine weitere Erhöhung der Zahl gepufferter Datenblöcke pro Arbeitsprozessor führt im zugrundeliegenden Beispiel zu Pufferüberläufen.

- 36) Ab welcher Mindestgröße für einen aufzubereitenden Hyper-Absatz sich solche einfachen Strategien "lohnend", kann mit in [Klei82] beschriebenen Methoden ausgerechnet werden.

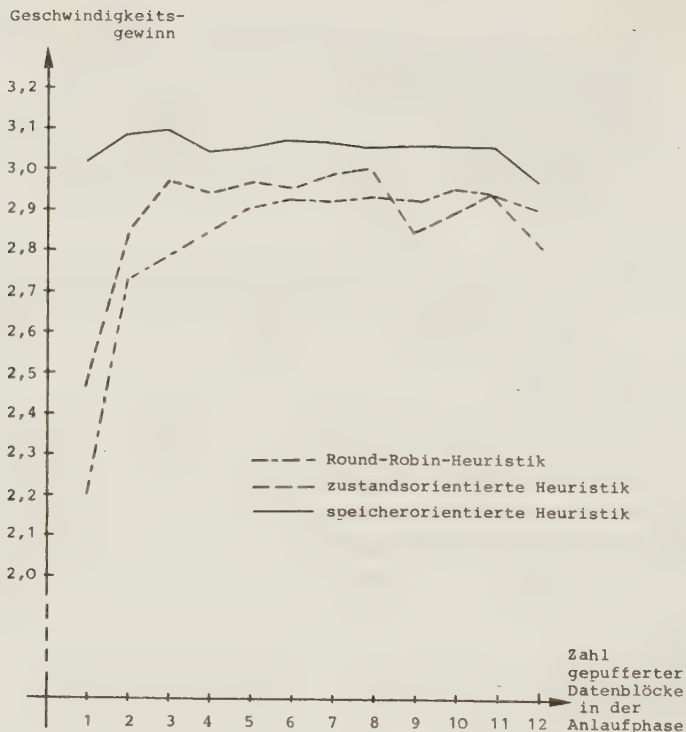


Abb. 4.7: Geschwindigkeitsgewinne für parallele Heuristiken zur Textformatierung auf der Basis von Hyper-Absätzen

Bei Verwendung der zustandsorientierten Heuristik wird der Punkt, ab dem keine kontinuierliche Steigerung des Geschwindigkeitsgewinns mehr erfolgt, bereits mit einer Zahl von drei gepufferten Datenblöcken pro Arbeitsprozessor erreicht. Von da ab zeichnet sich die Kurve gegenüber der der Round-Robin-Heuristik durch wesentlich stärkere Schwankungen aus.

Der Gebrauch der speicherorientierten Heuristik bringt von Anfang an die höchsten Geschwindigkeitsgewinne aller Heuristiken. Auch Schwankungen um den besten Wert für den Geschwindigkeitsgewinn sind bei dieser am geringsten.

Die in Abb. 4.7 dargestellte Meßreihe belegt offensichtlich, daß bei der zugrundeliegenden Verteilung des Eingabedatenstroms auf die Arbeitsprozessoren die speicherorientierte Heuristik die besten Ergebnisse liefert. Fragen nach den Ursachen lassen sich jedoch daraus noch nicht genau beantworten. Insbesondere bleiben folgende Fragen offen:

1. Wie läßt sich erklären, daß die speicherorientierte Heuristik in jedem Fall bessere Ergebnisse als die anderen Heuristiken liefert?
2. Wieso liegt für eine kleine Zahl gepufferter Datenblöcke der Geschwindigkeitsgewinn der Round-Robin-Heuristik schlechter als bei der zustandsorientierten Heuristik und für große Werte besser?
3. Wie kommt es zu den starken Schwankungen im Kurvenverlauf der zustandsorientierten Heuristik?

Antworten auf diese Fragen gibt eine Analyse der Wartezeiten, die im Programmablauf für die verschiedenen Heuristiken auf dem Kontrollprozessor auftreten. Eine Analyse der Zeitpunkte, zu denen gewartet wird und welcher der vier Arbeitsprozessoren dafür verantwortlich ist, ermittelte die in den nachfolgenden Punkten zusammengefaßten Resultate, die obige Fragen beantworten:

1. Weder mit Round-Robin-Heuristik noch mit zustandsorientierter Heuristik war es möglich, Wartephasen auf dem Kontrollprozessor auszuschließen. Dagegen ermöglichte es die Verwendung der speicherorientierten Heuristik in fast allen Fällen, das Programm ohne Wartezeiten auf dem Kontrollprozessor während des eigentlichen Laufs abzuschließen. (Lediglich in der Initialisierungsphase des Rahmens traten Wartezeiten auf.) Dies erklärt die guten Werte für den Geschwindigkeitsgewinn mit Gebrauch der speicherorientierten Heuristik gegenüber den anderen beiden Heuristiken.

2. Bei keinem Ablauf mit der Round-Robin-Heuristik war für weniger als acht gepufferte Datenblöcke pro Arbeitsprozessor ein Warten des Kontrollprozessors in der Kernphase vermeidbar. Bei Verwendung der zustandsorientierten Heuristik war es jedoch bereits bei Pufferung von drei Datenblöcken pro Arbeitsprozessor möglich, Wartezeiten auf dem Kontrollprozessor aus der Kernphase heraus in die Auslaufphase zu verlagern. Dies erklärt die Kurvenverläufe über den Geschwindigkeitsgewinn bei Round-Robin-Heuristik und zustandsorientierter Heuristik im Anfangsbereich.

3. Für eine große Zahl gepufferter Datenblöcke traten Wartezeiten auf dem Kontrollprozessor auf, die ausschließlich durch den vierten Arbeitsprozessor verursacht wurden, d. h. der vierte Arbeitsprozessor arbeitete scheinbar langsamer als die übrigen Arbeitsprozessoren. Dies erklärt sich aus der Strategie zur Bedienung von Systemaufrufen durch das EGPA-Betriebssystem, die die Arbeitsprozessoren unterschiedlich stark bevorzugt.³⁷⁾ Während die speicherorientierte Heuristik in der Lage ist, dies durch unterschiedlich starke Arbeitsbelastung der Arbeitsprozessoren auszugleichen, werden mit Gebrauch der anderen Heuristiken die zu Beginn des Kap. 3.2.4 begründeten Effekte bei der Parallelverarbeitung mit unterschiedlich schnellen Prozessoren deutlich.

Bei der zustandsorientierten Heuristik werden im Gegensatz zur Round-Robin-Heuristik je nach zufällig entstehender Reihenfolge von "Betriebssystem"-Ereignissen auf verschiedenen Prozessoren die Arbeitsprozessoren unterschiedlich stark mit Verarbeitungsaufgaben beauftragt. In der Konsequenz führt dies dazu, daß der Prozessor A_1 häufiger als der Prozessor A_4

37) Bei "gleichzeitigem" Auftreten von Ereignissen auf verschiedenen A-Prozessoren, welche Aktivitäten des Betriebssystems auslösen (Interrupts, Pagefaults), wird immer der A-Prozessor mit dem kleineren Index zuerst bedient. A_1 besitzt in einem solchen Fall eine höhere Priorität als A_2 , A_4 besitzt in solchen Fällen bei der Bedienung durch das Betriebssystem immer die niedrigste Priorität.

"bedient" wird und der Prozessor A_4 im Durchschnitt eine längere Zeit als der Prozessor A_1 für das Durchlaufen einer Betriebssystemroutine benötigt. (Diese Vermutung wurde durch Messungen bestätigt.) Die Folge ist, daß bei Verwendung der zustandsorientierten Heuristik für eine große Menge gepufferter Datenblöcke der Puffer des Prozessors A_1 häufig überzulaufen droht und sich dadurch zusätzliche Verzögerungen bei der Bearbeitung ergeben. Dies erklärt, warum für eine große Menge gepufferter Datenblöcke die zustandsorientierte Heuristik schlechtere Ergebnisse als die Round-Robin-Heuristik liefert.

Die starken Schwankungen im Kurvenverlauf der zustandsorientierten Heuristik erklären sich daraus, daß die Häufigkeit drohender Pufferüberläufe auf verschiedenen Prozessoren und dadurch bedingte Verzögerungen von der zufällig "günstigen" oder "ungünstigen" Abfolge von "großen" und "kleinen" Datenblöcken im Puffer eines Arbeitsprozessors abhängig ist. Bei der zustandsorientierten Heuristik wirken sich im Vergleich zu den anderen Heuristiken stark variierende Größen von Datenblöcken bei einer großen Menge gepufferter Daten auf die Gesamtverarbeitungszeit am deutlichsten aus.

4.2 Netzwerkanalyse

Das Programm zur Berechnung der Stromverteilung in einem elektrischen Netzwerk steht für eine der vielen im EGPA-System durchgeführten Implementierungen, die sich im wesentlichen auf Methoden der numerischen Mathematik abstützen. Eine Zusammenfassung von Algorithmus und Ergebnissen der (aus [Vas83] entwickelten) Implementierungen, soweit sie nicht in Kap. 4.2.2 beschrieben sind, wird in [HoVa84] gegeben. In [DDJT79] befindet sich eine Beschreibung einer Implementierung dieser Aufgabe auf CM*, in [GrTa82] dient dieses Beispiel als Grundlage eines Verfahrens zur theoretischen Bestimmung von Laufzeiten von Parallelprogrammen auf Multiprozessorkonfigurationen. In [Vas84] wird

ein Vergleich der entwickelten Parallel-Implementierung mit Implementierungen auf SMS und CM* durchgeführt.

Da die programmiersprachliche Formulierung des Kerns des Algorithmus nur ca. zwei Programmseiten umfaßt, erscheint die Vorgehensweise bei der Parallelisierung großer ordnungserhaltender Programmsysteme nicht notwendigerweise angebracht. Damit bietet dieses Beispiel jedoch eine gute Möglichkeit, die in den vergangenen Kapiteln beschriebene Methode mit "direkten" Methoden auch quantitativ zu vergleichen. Darüberhinaus läßt sich hier recht anschaulich darstellen, wie die anfallende Arbeit so auf das zugrundeliegende hierarchische Multiprozessorsystem aufgeteilt werden kann, daß alle Prozessoren (einschl. des Kontrollprozessors) zu fast 100 % ausgelastet sind.

Ausgangspunkt ist das in Abb. 4.8 gegebene Netzwerk. Eine Kette

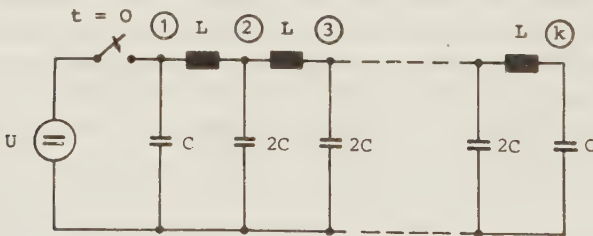


Abb. 4.8: Untersuchtes Netzwerk

von $(k-1)$ Abschnitten aus Kapazitäten und Induktivitäten (bekannter Größe), die eine Übertragungsstrecke simulieren, werde zum Zeitpunkt $t = 0$ mit einer Spannungsquelle verbunden. Gesucht seien die Spannungen an den k Knotenpunkten zu fest vorgegebenen Zeitpunkten. Abb. 4.9 zeigt den Spannungsverlauf in den Knoten 1, 3 und 8 eines aus acht Knoten bestehenden Netzwerkes über dreihundert Zeiteinheiten hinweg an, wobei die Größe einer Zeiteinheit 0.25 msec beträgt.

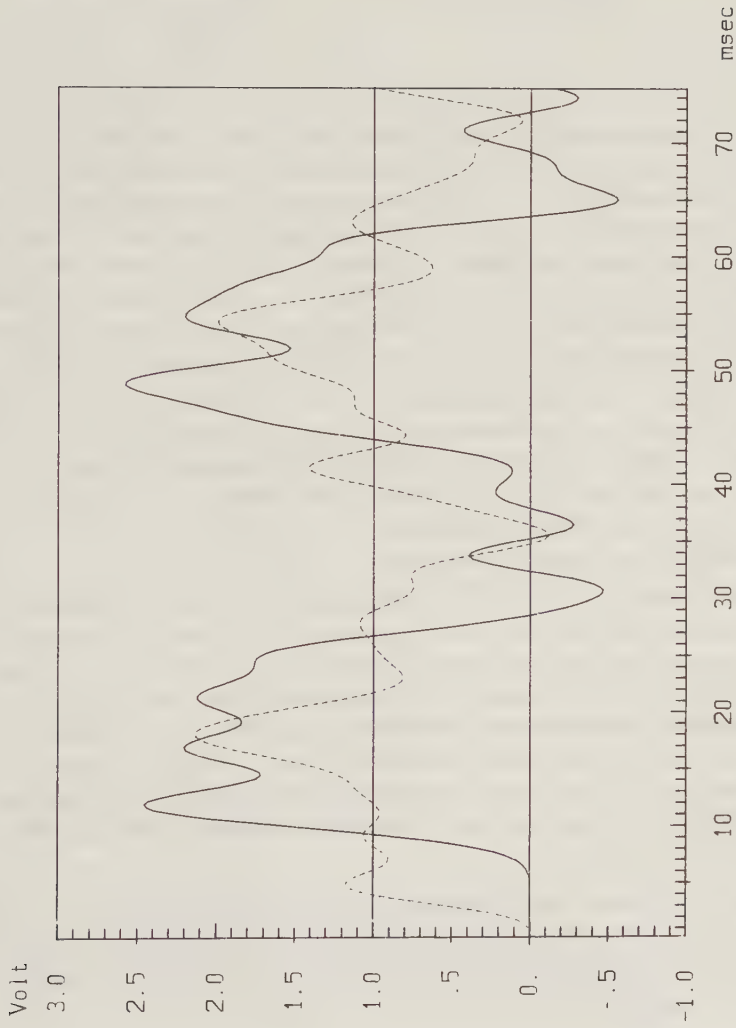


Abb. 4. 9: Spannungsverlauf in den Knoten 1, 3 (gestrichelt) und 8

Nachfolgend wird die Struktur des Programms zur Berechnung dieser Werte vorgestellt, daraus die Möglichkeiten zur Parallelisierung abgeleitet und ein Überblick über erzielte Ergebnisse gegeben.

4.2.1 Programmstruktur und Alternativen zur Parallelisierung

Für die Implementierung von [Vas83] wurde das das Netzwerk der Abb. 4.8 beschreibende Differentialgleichungssystem mit Hilfe einer Trapezmethode in ein System aus Differenzengleichungen übergeführt und die Spannungswerte an den verschiedenen Knoten zu den verschiedenen Zeitpunkten iterativ von den bekannten Anfangsbedingungen ausgehend berechnet. Diese Methode der Differenz-Admittanzen (eine ausführliche Beschreibung befindet sich in [HoVa84]) führt auf die Programmstruktur der Abb. 4.10.

Zur Analyse der Teilaufgaben, deren Parallelisierung einen Geschwindigkeitsgewinn erwarten läßt, wurden die in Abb. 4.10 mit UKCOMP, UZCOMP und HCOMP bezeichneten Teilaufgaben herangezogen. UKCOMP und UZCOMP sind Multiplikationen einer Matrix mit einem Vektor, HCOMP besteht aus linearen Transformationen von Vektoren, die elementweise durchführbar sind.

Eine Übersicht über die zeitlichen Anteile dieser Teilaufgaben an der Gesamtverweilzeit des Iterationsteils (vgl. Abb. 4.11) zeigt, daß aufgrund des Anteils UKCOMP³⁸⁾ an der gesamten Iteration eine Aufteilung von Funktionen auf Prozessoren als Möglichkeit zur Verringerung der Programmlaufzeit Parallelarbeit von vornherein ausscheidet. Dagegen kann bereits mit einer Vervielfachung der Funktion UKCOMP und einer Aufteilung der durch diese Funktion verwendeten Daten auf ein beliebig großes Multiprozessorsystem im

38) Der hohe Rechenzeitbedarf von UKCOMP gegenüber UZCOMP erklärt sich daraus, daß innerhalb von UZCOMP eine schwach besetzte Matrix mit einem Vektor multipliziert wird, sodaß im Gegensatz zu UKCOMP Techniken angewendet werden, bei der die von UZCOMP verwendete Matrix in Vektorform abgespeichert und innerhalb der Rechnung verwendet werden kann.

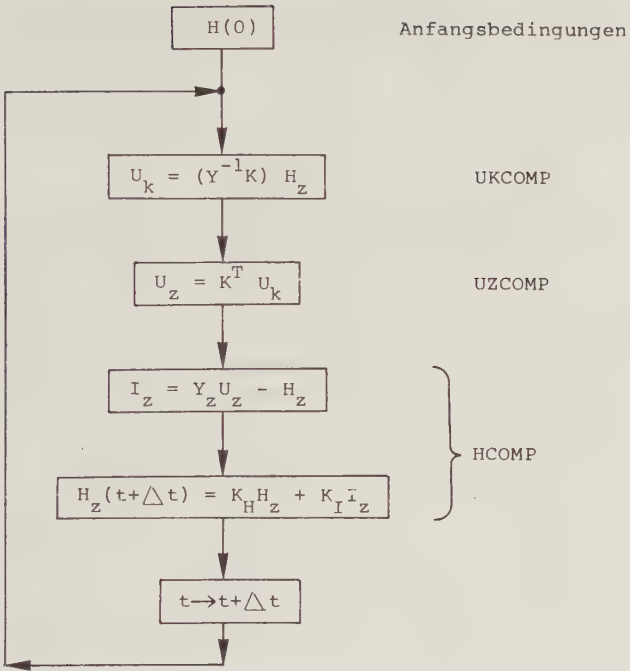


Abb. 4.10: Struktur des Programms zur Netzwerkanalyse

vorliegenden Fall mit einem fast zwanzigfachen Geschwindigkeitsgewinn³⁹⁾ gegenüber dem sequentiellen Programm gerechnet werden, was den Einsatz eines Multiprozessorsystems auf jeden Fall als lohnend erscheinen läßt.

Modul	Laufzeit in Sekunden	Laufzeit in Prozent
UKCOMP	554.8	94.7
UZCOMP	19.4	3.3
HCOMP	9.7	1.7
andere	1.9	0.3

Abb. 4.11: Laufzeit der Funktionen für die Netzwerkanalyse am Beispiel eines Netzwerks von 100 Knoten für 300 Iterationen

Als Grundlage zur Aufteilung der Datenmenge sei die Abb. 4.12 betrachtet, die eine schematische Darstellung der von den einzelnen Teilaufgaben verwendeten Daten zeigt.

Zur Form der Aufteilung von Matrizen und Vektoren bieten sich zwei Möglichkeiten an, eine Aufteilung in der Dimension k und eine in der Dimension z .

Bei einer Aufteilung in der Dimension k in p Teile (für p Arbeitsprozessoren) kann in UKCOMP jeder der Arbeitsprozessoren den p -ten Teil der Komponenten von UK berechnen. Dazu wäre es jedoch erforderlich, daß jeder Arbeitsprozessor eine Kopie des gesamten H-Vektors haben müßte.

Bei einer Aufteilung in der Dimension z (wie sie in Abb. 4.12 für einen Wert von $p=2$ skizziert ist) können die Vektoren der Dimension z in p Teile geteilt werden. In diesem Fall muß jeder Arbeitsprozessor Platz für alle Komponenten von UK vorsehen und berechnet innerhalb von UKCOMP nur den p -ten Anteil des neuen UK.

- 39) Bei Verwendung eines hierarchischen Multiprozessorsystems mit weniger als zwanzig Arbeitsprozessoren (wie etwa dem zugrundeliegenden EGPA-System) kann mit dieser Aufteilung ein Geschwindigkeitsgewinn erwartet werden, der etwa der Zahl der Arbeitsprozessoren entspricht. Diese kann dann noch gesteigert werden, wenn der Kontrollprozessor an der Berechnung von UK beteiligt wird.

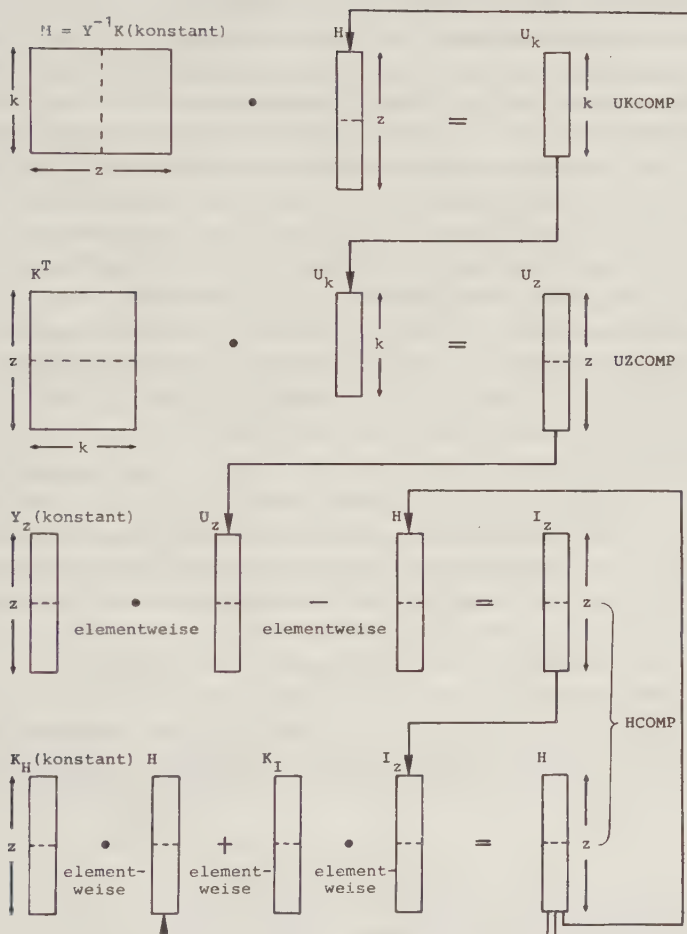


Abb. 4.12: Datenflußstruktur des Programms zur Netzwerkanalyse in Form von Matrizen und Vektoren

Der Kontrollprozessor muß in diesem Fall die von den Arbeitsprozessoren angelieferten UK-Anteile aufaddieren.⁴⁰⁾

Innerhalb der durchgeführten Implementierungen wurde von einer Aufteilung der Daten in der z-Dimension ausgegangen, da es diese auf einfache Weise erlaubt, neben UKCOMP die Teilaufgabe HCOMP zusätzlich parallel auf den Arbeitsprozessoren auszuführen, ohne daß Daten mehrfach an verschiedene Prozessoren verteilt werden müssen. Die Beachtung dieses Gesichtspunktes ist für das der Implementierung zugrundeliegende EGPA-System von Bedeutung, da in EGPA keine Routinen vorgesehen wurden, Daten (per Broadcast) gleichzeitig an alle Arbeitsprozessoren zu verteilen.

Mit dieser Aufteilung von Daten und Funktionen auf die Prozessoren des hierarchischen Multiprozessorsystems (Aufsummation der UK-Anteile und UZCOMP auf dem Kontrollprozessor, HCOMP und UKCOMP auf den Arbeitsprozessoren) sind nun zwei Varianten zur Ablaufsteuerung möglich.

In Variante 1 (eingabeorientiert bezgl. UK) wird ein neues UZ und ein neues H komponentenweise berechnet, ein neues UK anteilsweise.⁴¹⁾ In diesem Fall wird in UKCOMP über eine "längere" Zeitspanne hinweg nur jeweils eine Komponente von H bzw. UZ verwendet, sodaß die Berechnung eines neuen UK sinnvoll bereits dann begonnen werden kann, wenn lediglich eine Komponente von UZ vorliegt. Der Nachteil liegt darin, daß zur Berechnung einer Komponente eines neuen UZ der Kontrollprozessor den gesamten neu berechneten Vektor UK benötigt, darauf also unter Umständen recht lange warten muß.

40) Bei einer mehrstufigen Hierarchie der Prozesselemente im Multiprozessorsystem (z. B. als Baum aus EGPA-Elementarpyramiden [FKLV83]) kann diese Addition in Form einer Baumhöhenreduktion wie in Kap. 2.1.2 skizziert durchgeführt werden.

In Variante 2 (ausgabeorientiert bezgl. UK) wird ein neues UZ und ein neues H anteilsweise, ein neues UK komponentenweise berechnet. Dies hat den Vorteil, daß mit der Berechnung eines neuen UZ-Anteils schon dann begonnen werden kann, wenn bereits eine neu berechnete Komponente von UK von allen Arbeitsprozessoren vorliegt. Nachteil ist, daß die Berechnung von H und UK erst dann erfolgen kann, wenn das gesamte neue UZ vorliegt.

Der zweiten Variante wurde jedoch deshalb der Vorzug gegeben, da - wie Messungen ergeben haben - das Warten auf ein neues UZ durch die Arbeitsprozessoren aufgrund der Laufzeiten der verschiedenen Teilaufgaben und der möglichen Überlappung bei der Berechnung von UK und UZ weniger lang dauert als das (in Variante 1 erforderliche) Warten auf ein neues UK durch den Kontrollprozessor.

Dies mag als Beispiel dafür dienen, wie bei der Aufteilung von Vektoren und Matrizen auf Prozessoren eines Multiprozessorsystems vorgegangen werden kann und welche Überlegungen dabei eine Rolle spielen. Ergebnisse durchgeführter Implementierungen (Kap. 4.2.2 berücksichtigt Aufteilungen in der z-Dimension mit einer Bearbeitung nach Variante 2) haben die Richtigkeit der angestellten Überlegungen bestätigt.

- 41) Unter der komponentenweisen Berechnung von UK soll eine Berechnung innerhalb einer Laufschleife der Form
- ```

FOR LK TO K
DO UK[LK] := 0;
 FOR LZ TO Z
 DO UK[LK] := UK[LK] + M[LK,LZ] * H[LZ]
 OD
OD

```
- , unter der anteilsweisen Berechnung von UK eine Berechnung innerhalb einer Laufschleife der Form
- ```

FOR LK TO K DO UK[LK] := 0 OD;
FOR LZ TO Z
DO FOR LK TO K
  DO UK[LK] := UK[LK] + M[LK,LZ] * H[LZ]
OD
OD

```
- verstanden werden.

4.2.2 Ergebnisse von Parallelisierungsstrategien

Bei durchgeführten Implementierungen mit den in Kap. 4.2.1 erläuterten Voraussetzungen wurde wie in [DDJT79] von einer Netzwerkgröße von 100 Knoten ausgegangen, über die 300 Iterationen ausgeführt wurden. Bei der Aufteilung der zu verarbeitenden Daten in der z-Dimension wurde dem Kontrollprozessor ein zwischen 0 und 20 % variierender Anteil gegeben und der Rest gleichmäßig unter den Arbeitsprozessoren aufgeteilt.

Neben einer Implementierung mit den "E/A-orientierten" Methoden des Kap. 2 (in diesem Fall Transferieren berechneter UZ- und UK-Komponenten über Kommunikationsanweisungen) wurde eine Implementierung mit einer "direkten" Methode durchgeführt. Bei letzterer wurde über Werte von Zustandsvariablen angezeigt, wann eine neue UK-Komponente bzw. ein neuer UZ-Vektor errechnet worden war (zur Gestaltung solcher Zustandsanzeigen und damit in Verbindung stehender Fragen der Synchronisation vgl. Kap. 5.2).

Abb. 4.13 zeigt in Abhängigkeit des dem Kontrollprozessor zugeordneten Anteils die erreichten Werte für den Geschwindigkeitsgewinn gegenüber dem sequentiellen Programm.

Die Ergebnisse lassen sich in folgenden Punkten zusammenfassen:

1. Die erzielten Geschwindigkeitsgewinne entsprechen den Erwartungen. Zeitverluste durch organisatorische Maßnahmen spielen auch in parallelen Implementierungen praktisch keine Rolle. Schon dann, wenn der Kontrollprozessor nicht am eigentlich zeitintensiven Teil des Programms (Teilaufgabe UKCOMP) beteiligt wird, ist ein Geschwindigkeitsgewinn von knapp über 4 erreichbar.

Dies erklärt sich daraus, daß wegen des überlappten Arbeitens von Kontroll- und Arbeitsprozessoren der maximal mögliche Geschwindigkeitsgewinn aufgrund der Verarbeitungszeiten fast vollständig durch den Arbeitsanteil bestimmt wird, den ein Arbeitsprozessor zu bewältigen hat. Dieser ist jedoch kleiner als ein Viertel der Gesamtbearbeitungszeit, da im Parallelpro-

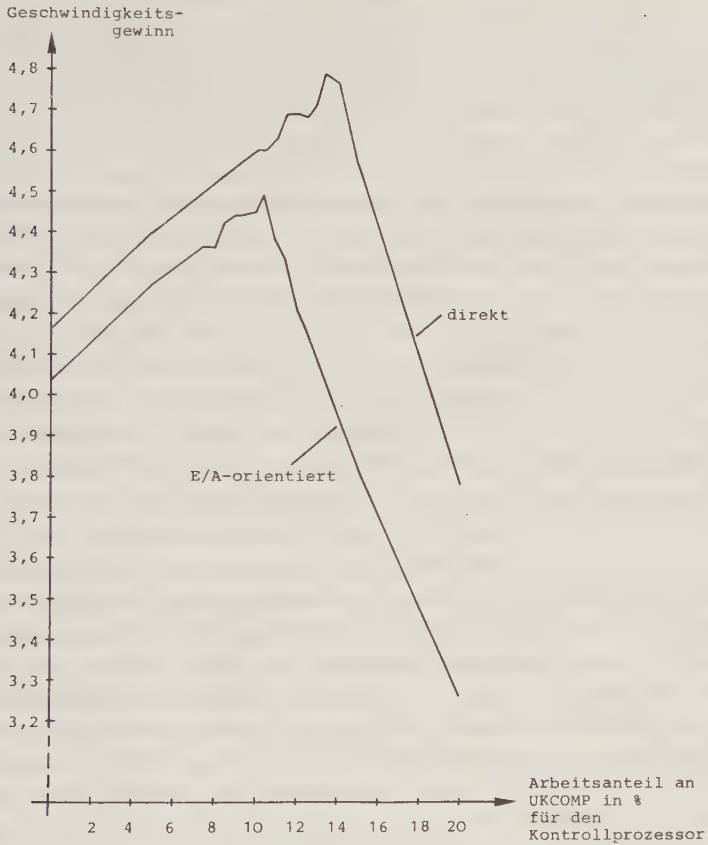


Abb. 4.13: Geschwindigkeitsgewinne für parallele Implementierungen der Netzwerkanalyse mit verschiedenen Methoden

gramm auch der Kontrollprozessor an der Gesamtarbeit beteiligt ist.

2. Die mit der "E/A-orientierten" Methode durchgeführte Implementierung erreicht die besten Werte für den Geschwindigkeitsgewinn wenn etwa 10 % der rechenintensiven Arbeiten dem Kontrollprozessor zugeordnet werden, die mit der "direkten" Methode durchgeführte Implementierung erreicht diese bei einer Zuordnung von etwa 15 %. Dies läßt sich mit dem erhöhten Synchronisationsaufwand bei der "E/A-orientierten" Methode erklären, der dem Kontrollprozessor etwas mehr von der Zeit wegnimmt, die bei der "direkten" Methode für die Rechnungen übrigbleibt. Während bei der "E/A-orientierten" Methode für jede neu berechnete und über Kommunikationsanweisung transferierte UZ-Komponente Synchronisationen (innerhalb der Kommunikationsanweisungen) erforderlich werden, geschieht dies bei der "direkten" Methode (wie oben skizziert) nur einmal.
3. Implementierungen mit der "direkten" Methode haben eine um etwa 6 % kürzere Laufzeit als Implementierungen mit der "E/A-orientierten" Methode. Dies zeigt, daß für dieses Beispiel die Verwendung der "E/A-orientierten" Methode eine durchaus akzeptabel geringe Verschlechterung der Effizienz gegenüber der "direkten" Methode hat. Bei Beispielen, bei denen die "direkte" Methode keine Einsparung an erforderlichen Synchronisationen bringt (wie z. B. bei der Textformatierung) dürften die Unterschiede noch geringer sein, ebenso dann, wenn eine "besser" gewichtete Verteilung der Arbeit unter den Arbeitsprozessoren erfolgt.

Zu berücksichtigen ist aber, daß die im dritten Punkt kommentierten geringen Unterschiede zwischen der "E/A-orientierten" Methode und der "direkten" Methode darauf zurückzuführen sind, daß die Rechenzeiten gegenüber den Transport- und Synchronisationszeiten deutlich überwiegen. (Der Anteil "organisatorischer" Maßnahmen an der Gesamtbearbeitungszeit liegt im Parallelprogramm unter 10 %.)

Wenn geringere Unterschiede zwischen Rechen- und "Verwaltungszeiten" bestehen, werden die Unterschiede im erreichbaren Geschwindigkeitsgewinn zwischen "E/A-orientierten" und "direkten" Implementierungen größer, da dann die längere Kommunikationszeit der "E/A-orientierten" Implementierung stärker die Gesamtausführungszeit beeinflusst.

Bei "großen" ordnungserhaltenden Programmsystemen wie Textformatierprogrammen und Compilern spielen derartige Überlegungen praktisch keine Rolle, da aus Gründen der Transparenz im allgemeinen bereits im sequentiellen Programm E/A-orientierte Kommunikationsformen implementiert werden.

Dies zeigt, daß die E/A-orientierte Parallelisierungsmethode für ordnungserhaltende Programmsysteme "zugeschnitten" ist und in ordnungsfreien Programmsystemen anwendbar ist. Es zeigt aber auch, daß in Systemen, in denen "direkte" Implementierungen möglich sind, die "E/A-orientierte" Implementierungsmethode aufgrund schlechterer Effizienz mehr die Rolle einer Übergangslösung haben sollte.

Die "E/A-orientierte" Implementierung hat jedoch auch dort eine Berechtigung, da sie (vgl. Kap. 5) ein "systemgesteuertes" Testen von Programmen erlaubt und mit geringerem Aufwand als die direkte Implementierung von speichergekoppelten Rechnersystemen auf botschaftsgekoppelte Rechnersysteme übertragen werden kann. Zudem sind "direkte" Implementierungen sinnvoll nur in Systemen ausführbar, die - wie EGPA - über gemeinsam benutzbare Speicher gekoppelt sind (vgl. Kap. 6.2).

Abschließend sei noch eine Implementierung mit einer Nachbarhilfe zwischen Kontrollprozessor und Arbeitsprozessoren erwähnt. Diese wird in der Form gestaltet, daß nach gleichmäßiger Aufteilung der von den Arbeitsprozessoren zu verarbeitenden Eingabedaten und erforderlichen organisatorischen Arbeiten der Kontrollprozessor den Arbeitsprozessoren bei der Berechnung von UK-Komponenten hilft, die mit ihrer Arbeit am weitesten zurückliegen.

Die ordnungsfreie Struktur des Programmteils zur Berechnung der UK-Komponenten (diese können nach Vorliegen des Vektors H in einer beliebigen Reihenfolge berechnet werden) erlaubt es, daß im Fall der Nachbarhilfe etwa der Kontrollprozessor die UK-Komponenten für einen Arbeitsprozessor von "hinten" her (Reihenfolge $UK_k, UK_{k-1}, \dots$) berechnet, während der Arbeitsprozessor seinen Teil von "vorne" (Reihenfolge $UK_1, UK_2, \dots$) her bearbeitet. Wenn Teilergebnisse der Rechnung auf prozessorprivaten Daten gespeichert werden und nur das letztendlich erzielte Ergebnis per Zuweisung im von beiden beteiligten Prozessoren zugreifbaren Bereich abgelegt wird, ist es sogar tolerierbar, einzelne UK-Komponenten doppelt zu berechnen. Die Nachbarhilfe zwischen Prozessoren erlaubt es, dynamisch während des Programmlaufs die vorhandene Arbeit fast optimal unter den beteiligten Prozessoren aufzuteilen.

Das Prinzip der Nachbarhilfe ist im EGPA-System in vielen Anwendungen realisiert worden, um die Arbeitsprozessoren möglichst gleichmäßig mit Arbeit auszulasten und dadurch einen höheren Geschwindigkeitsgewinn zu erreichen. In [Fri84] werden die Auswirkungen der Nachbarhilfe auf die unterschiedliche Arbeitsauslastung von Arbeitsprozessoren am Beispiel der Berechnung von Minima mehrdimensionaler Funktionen ausgeführt.

Das Prinzip der Nachbarhilfe bei der Bearbeitung von Aufgaben durch mehrere Prozessoren ist verwandt mit den in [Kung76] beschriebenen adaptiven Implementierungen (vgl. auch Kap. 3.3). Es unterscheidet sich jedoch darin, daß bei der Nachbarhilfe direkt (über die Speicherkopplung) auf die Daten fremder Prozessoren innerhalb einer Aufgabe zugegriffen wird, während adaptive Implementierungen mehr den Gedanken in den Vordergrund stellen, daß sich die Zusammenarbeit zwischen verschiedenen Prozessoren auf die Abarbeitung eines Stroms eigenständiger Aufgaben bezieht.

In Abb. 4.14 sind für die Netzwerkanalyse die erreichten Geschwindigkeitsgewinne der Implementierung mit Nachbarhilfe gegenüber dem sequentiellen Programm⁴²⁾ in Abhängigkeit der Zahl verwendeter Arbeitsprozessoren den Ergebnissen einer Parallelimplementierung gegenüberstellt, bei der der Kontrollprozessor nicht an der von den Arbeitsprozessoren durchgeführten Arbeit (d. h. der Berechnung der UK-Komponenten) beteiligt ist.

Zahl verfügbarer Arbeitsprozessoren	Geschwindigkeitsgewinn bei Implementierung ohne Nachbarhilfe	Implementierung mit Nachbarhilfe	Verbesserung durch Nachbarhilfe
2	2.15	2.83	32 %
3	3.14	3.61	15 %
4	4.04	4.37	8 %

Abb. 4.14: Geschwindigkeitsgewinne für Implementierungen mit und ohne Nachbarhilfe in Abhängigkeit der Zahl verfügbarer Arbeitsprozessoren

Die Ergebnisse zeigen, daß die Nachbarhilfe zwischen Prozessoren (als Form einer speicherorientierten Heuristik) umso attraktiver ist, je weniger Prozessoren das zugrundeliegende Multiprozessor-system besitzt und daß mit wachsender Zahl verfügbarer Ressourcen der "Implementierungsstil" immer weniger von Bedeutung ist.

⁴²⁾ Die mit der Implementierung mit Nachbarhilfe erzielten Gewinne liegen deshalb schlechter als die der Abb. 4.13, da der Kontrollprozessor nicht an der Teilaufgabe HCOMP beteiligt wurde, sondern erst dann mit seiner Hilfe eingriff, wenn H vollständig neu berechnet war. Eine zusätzliche Synchronisation zur Ermittlung des Fertigstellungszeitpunktes des Vektors H durch den Kontrollprozessor und die häufig erforderliche Bestimmung des am weitesten mit seiner Arbeit zurückliegenden Arbeitsprozessors sind weitere Ursachen, die die Verschlechterung erklären.

Dies ist dann wichtig, wenn die Parallelprogrammierung auf Systemen durchgeführt wird, die keine so weitreichenden Möglichkeiten zur Kontrolle von Programmen auf fremden Prozessoren bieten, wie sie für die Erstellung von Parallelimplementierungen unter Verwendung von Prinzipien der Nachbarhilfe eigentlich notwendig wären. So ist zur Realisierbarkeit der Nachbarhilfe eine Speicherkopplung der einander helfenden Prozessoren praktisch unerlässlich (vgl. auch Kap. 6.2).

Für Systeme mit stark eingeschränkten Ressourcen, in denen Implementierungen mit Nachbarhilfe nicht möglich oder nicht sinnvoll sind, wird zum Auffinden von zufriedenstellenden Parallelimplementierungen ein sicher stärkerer Aufwand in die theoretische Analyse zur Bestimmung "guter" Formen der Arbeitsaufteilung unter den Prozessoren des betrachteten Multiprozessorsystems zu investieren sein.

Die Bewertung solcher Fragen und die dann folgende Umsetzung von sequentiellen Programmen in zufriedenstellende Parallelimplementierungen lassen sich sicher leichter durchführen, wenn ein systematischer Parallelisierungsansatz in Verbindung mit einer auf das System geeignet zugeschnittenen parallelen Programmierungsumgebung als Grundlage verwendet wird. Welche Fragen davon berührt werden und inwieweit besondere Fragen paralleler Implementierungen durch ein "System" abgehandelt werden können, ist Hauptthema des nachfolgenden Abschnitts "Implementierungsaspekte".

5 Implementierungsaspekte

In Kap. 3 beschriebene parallele Verarbeitungsstrategien für ordnungserhaltende Programmsysteme wurden verbal formuliert, d. h. ohne Rücksicht darauf, wie die Strategien auf einem real vorhandenen System zu programmieren sind. Dabei mußten einige Fragen nach speziellen Problemen auf der Seite der Parallelimplementierung zwangsweise offen bleiben. Nachfolgend werden einige der Fragen, die die Erstellung von Parallelprogrammen (im Sinn der Def. 2.5 des Kap. 2.2) betreffen, näher untersucht.

Grundlage der in Kap. 4 beschriebenen Parallel-Implementierungen sind drei verschiedene Programmierungsumgebungen, deren Struktur in Kap. 5.1 vorgestellt wird.

Anhand der beschriebenen Hilfsmittel zur Parallelprogrammierung wird auf Fragen der Kommunikation und Synchronisation zwischen Prozessoren eines Multiprozessorsystems (Kap. 5.2) und auf Fragen der Behandlung von Laufzeitfehlern und daraus resultierenden Systemverklemmungen (Kap. 5.3) eingegangen.

In Kap. 5.4 werden schließlich Methoden zusammengestellt, Parallelprogramme auf Korrektheit zu testen bzw. Fehler in Parallelprogrammen zu finden. Gerade der zuletzt genannte Aspekt, in Veröffentlichungen zu Parallel-Implementierungen gewöhnlich nicht oder nur am Rande behandelt, zeigt, wie wichtig eine Schnittstelle zwischen Benutzerprogramm und Betriebssystem zur Unterstützung des Anwenders bei der Erstellung von Parallelprogrammen ist und welche Vorzüge die in Kap. 2 beschriebene Vorgehensweise bei der Implementierung von Programmen auf Multiprozessorsystemen bietet.

5.1 Struktur zugrundeliegender Programmierumgebungen

Hauptgrund zur Definition von Hilfsmitteln zur Parallelprogrammierung ist, dem Benutzer die Erlernung systemspezifischer Details des Multiprozessorsystems weitgehend zu ersparen und bei der Entwicklung von Parallelprogrammen zu unterstützen. Dies wird mit den hier beschriebenen Hilfsmitteln erreicht, indem Standardaufgaben innerhalb von Parallelprogrammen über ein für den Benutzer verfügbares Programmpaket abgewickelt werden.

Die hier vorgestellten Hilfsmittel für die Parallelprogrammierung auf einem EGPA-ähnlichen System gehen davon aus, daß jedem Prozessor ein Prozeß zugeordnet ist, eine Technik, wie sie auch bei den meisten auf CM* durchgeführten Implementierungen (vgl. etwa [Dem82], [Jon80]) Anwendung findet. Prozessorspezifische Benutzerprogramme werden als Unterprogramme formuliert. Bei der Erzeugung eines auf dem Multiprozessorsystem lauffähigen Ladeobjekts werden diese Unterprogramme mit einem Programmrahmen zusammengebunden.

Funktionen des Programmrahmens sind:

1. Initialisierung der Prozesse auf den einzelnen Prozessoren (Prozeßstart und Verlagerung der Prozesse auf die Prozessoren mit Hilfe von Diensten des Betriebssystems);
2. Bereitstellung von Funktionen zur Kontrolle von Aufgaben auf fremden Prozessoren (Starten von Aufgaben auf fremden Prozessoren, Warten auf die Beendigung von Aufgaben auf fremden Prozessoren, Auskunftsfunktionen über den Bearbeitungsstand von Aufgaben auf fremden Prozessoren);
3. Korrektes Beenden von Prozessen auf einzelnen Prozessoren.

Die anschließend kurz beschriebenen Hilfsmittel unterscheiden sich hauptsächlich darin, wie dem Benutzer Steuerungs- und Auskunftsfunktionen zur Verfügung gestellt werden und welche Möglichkeiten damit bestehen, in den Lauf von Programmen auf verschiedenen Prozessoren eines Multiprozessorsystems einzugreifen.

5.1.1 Elementare Hilfsmittel in einer EGPA-Umgebung

Der innerhalb des EGPA-Systems den Benutzern zur Verfügung gestellte Rahmen zur Erstellung von Parallelprogrammen ist maßgeblich von den Möglichkeiten beeinflusst worden, die das Betriebssystem und die Programmiersprache SL3 verfügbar gemacht haben.

Auf der Seite der verfügbaren Betriebssystemdienste gehören dazu Funktionen zum Starten und Stoppen von Prozessen, zum Senden und Empfangen von Botschaften zwischen verschiedenen Prozessen, auf der Seite der Programmiersprache die Möglichkeit, über Deklaration von Variablen innerhalb teilsystemeindeutiger Segmente direkt auf Daten fremder Prozesse zugreifen zu können.

Die Programmierumgebung selbst beinhaltet ein Programmpaket, welches im wesentlichen Routinen zur Steuerung von Aufgaben auf fremden Prozessoren enthält. In Anlehnung an die hierarchische Struktur der zugrundeliegenden Hardware (vgl. Abb. 1.2) ist ein Starten benutzerspezifizierter Aufgaben nur vom B-Prozessor (Kontrollprozessor) für die A-Prozessoren (Arbeitsprozessoren) möglich. Die Beeinflussung von Programmläufen unter den Arbeitsprozessoren erfolgt über die Manipulation gemeinsam zugreifbarer Daten.

Das Benutzerprogramm auf dem B-Prozessor wird als Prozedur (mit einem festgelegten Namen) formuliert, die vom Programmrahmen nach Durchführung der Initialisierungsarbeiten aufgerufen wird. Die Benutzerprogramme auf den A-Prozessoren (ebenfalls in Form von Prozeduren formuliert) werden vom Programmrahmen gestartet, wenn auf dem B-Prozessor ein Prozeduraufruf der Form

"Führe Unterprogramm i auf Prozessor A_j aus"

abgesetzt wurde. Nach diesem Aufruf wird auf Prozessor A_j das Unterprogramm i gestartet und läuft dort dann parallel zum Benutzerprogramm auf dem B-Prozessor weiter. Auf diese Weise können mehrere Programme hintereinander auf den A-Prozessoren gestartet werden, die von diesen in der Reihenfolge ihres Eintreffens bearbeitet werden.

Weitere von Programmrahmen zur Verfügung gestellte Aufrufe sind:

"Warte auf Beendigung der zuerst vergebenen Aufgabe auf Prozessor A_j bzw. auf allen Prozessoren" und

"Ermittle Auftragsbestand (Zahl vergebener Aufgaben, Zahl abgeschlossener Aufgaben) auf Prozessor A_j bzw. auf allen Prozessoren"

Der Informationsaustausch zwischen verschiedenen Prozessoren erfolgt direkt über Wertzuweisungen an Variable, die in gemeinsam genutzten Speichersegmenten vereinbart werden.

In [Zisch83] wird die Schnittstelle und die darauf aufbauende Programmierung für das System der ECPA-Pilotpyramide, in [Rat83] die Erweiterung für beliebig große Pyramidenstrukturen beschrieben.

Im Rahmen selbst sind Diagnosefunktionen implementiert, mit Hilfe derer der Benutzer einen Einblick in das Verhalten "seines" Parallelprogramms auf dem System gewinnen kann und welche im Fall von Programmfehlern schon recht weitgehende Analysen erlauben (vgl. Kap. 5.4). Insbesondere ist es zu Testzwecken möglich, Parallelprogramme für das Multiprozessorsystem oder zur Simulation auf dem Einzelprozessor (vgl. Kap. 5.4.2) zu erstellen, die sich auf der Ebene des Benutzerprogramms nicht unterscheiden.

Die Verwendung von Routinen des Programmpakets zur Synchronisation erlaubt die Erkennung und Beseitigung von Systemverklemmungen (vgl. Kap. 5.3). Eine inhaltsorientierte Spezifikation der Ursache einer Systemverklemmung kann ein solches Programmpaket jedoch nur bedingt leisten. Da das System den direkten Zugriff auf Daten fremder Prozessoren erlaubt, kann eine Synchronisation zwischen Prozessen über zyklisches Abfragen von Variableninhalten ohne "Umweg" über das Programmpaket erfolgen (benutzerimplementiertes Aktives Warten). Hierdurch hervorgerufene Systemverklemmungen sind dann nicht mehr analysierbar.

Die Erfahrungen mit diesem System haben maßgeblich zur Entwicklung der anderen Programmierschnittstellen und der in Kap. 6 systemdefinierten Schnittstelle auf der Basis von Kommunikationsprozeduren zum Nachrichtenaustausch zwischen Prozessoren beigetragen.

5.1.2 Elementare Hilfsmittel in einer UNIX⁴³⁾-Umgebung

Das UNIX-Betriebssystem [RiTh74] in Verbindung mit der Programmiersprache C [KeRi78] stellt wie EGPA dem Anwender eine Schnittstelle zur Verfügung, die eine recht systemnahe Programmierung erlaubt. Dies betrifft insbesondere die Möglichkeiten der Steuerung von Prozessen und der Implementierung von E/A-Systemen.

Der fundamentale Unterschied der innerhalb von UNIX erzeugbaren Prozeßsysteme zu denen, die innerhalb von EGPA gesteuert werden können, ist, daß im Basissystem⁴⁴⁾ keine Kommunikation verschiedener Prozesse über gemeinsam erreichbare Datenbereiche per Wertzuweisung oder direkter Zustandsabfragen möglich ist.

Kommunikation zwischen verschiedenen Prozessen im UNIX-System erfolgt über gemeinsam genutzte Files, Signale oder Pipes. Während Signale ungeeignet sind, spezifizierte Information zwischen Prozessen zu übertragen (mit Hilfe von Signalen ist es lediglich möglich, das Eintreffen von Ereignissen anzuzeigen), müssen die beiden anderen Möglichkeiten zur Prozessorkommunikation als Grundlage jeglicher Implementierung zur Realisierung von Parallelprogrammen innerhalb einer UNIX-Umgebung angesehen werden (vgl. hierzu auch [BlMS83]). Die Verwendung von gemeinsam genutzten Files oder Pipes⁴⁵⁾ für die Kommunikation zwischen

43) UNIX ist ein Warenzeichen von Bell Laboratories.

44) Auf UNIX-Erweiterungen im Hinblick auf die Realisierung von Multiprozessorsystemen [Lin81] soll in diesem Rahmen nicht weiter eingegangen werden.

verschiedenen Prozessen führt zwangsläufig auf die Formulierung von Routinen zum Informationsaustausch mittels Kommunikationsprozeduren und kommt damit der "E/A-orientierten" Methode zur Konstruktion von Parallelprogrammen aus sequentiellen Programmen stark entgegen. Innerhalb des für den Benutzer verfügbaren Rahmens zur Erstellung von Parallelprogrammen kommt die Implementierungsart (als Pipes) nicht zum Vorschein.

Wie im EGPA-System werden Benutzerprogramme als Unterprogramme formuliert, an die zum Zeitpunkt der Erzeugung des Lademoduls des Parallelprogramms ein Rahmen gebunden wird, innerhalb dessen die Prozeßsteuerung abgewickelt wird. Wie dort haben die Steuerprogramme auf den verschiedenen Prozessoren festgelegte Namen und werden vom Rahmen nach Durchführung der Initialisierungsarbeiten aufgerufen.

Mit Aufruf der Steuerprogramme werden den Benutzer Routinen Informationen über die Kennung des eigenen Prozessors und der erreichbaren Nachbarprozessoren übermittelt, welche bei der Kommunikation zwischen verschiedenen Prozessoren benötigt werden.

Zur eigentlichen Prozessorkommunikation stehen (in Anbetracht der Tatsache, daß Kommunikation nur mittels des über E/A-Primitiva realisierten Botschaftsmechanismus möglich ist) zwei Routinen zur Verfügung:

"Senddata" dient zum Übermitteln eines Informationsblocks (als Botschaft) an einen fremden Prozeß,

- 45) Pipes sind E/A-Kanäle, die zwischen kooperierenden Prozessen verwendet werden können. Ein Prozeß kann (mit Hilfe von E/A-Primitiva) in die Pipe Information (in Form eines Bytestroms) eintragen, die vom anderen Prozeß gelesen werden kann. Das System sorgt für die Pufferung der Daten und die Synchronisierung der Prozesse.

"Receivedata" zum Empfangen eines Informationsblockes von einem fremden Prozeß.

Für jeden Prozeß ist innerhalb des Programmrahmens ein Pufferbereich vorgesehen, in dem verschiedenen Prozessen zugängliche Information abgelegt wird. Dieser Bereich wird mit den Funktionen "Senddata" und "Receivedata" angesprochen.

Mittels der Routinen "Delay" und "Continue" (vgl. Kap. 5.3.4) wird der Beginn bzw. das Ende eines Zustandes angezeigt, in dem für den betreffenden Prozeß momentan aufgrund einer nicht erfüllten Bedingung ein Programmfortschritt nicht möglich ist.

Die Schnittstelle - eine detaillierte Beschreibung ist auf Anforderung verfügbar - ist in Form eines Simulationsprogramms (vgl. Kap. 5.4.2) implementiert. Sie dient im Rahmen dieser Ausführungen als Hintergrund zum Vergleich von Multiprozessorsystemen, die Kommunikation zwischen verschiedenen Prozessoren lediglich auf der Basis von Botschaften erlauben, mit Systemen, die zwischen Prozessoren direkt gemeinsam zugreifbare Datenbereiche zulassen.

5.1.3 Makro-Datenfluß

Wesentliche Kennzeichen der in den Kap. 5.1.1 und 5.1.2 beschriebenen elementaren Hilfsmittel für die Parallelprogrammierung ist, daß durch deren Gebrauch keine Erweiterung der Programmiersprache bzw. deren Compiler stattfindet. Damit ist der Benutzer an die durch die Sprache festgelegten Konstrukte bei der Programmierung gebunden. Wenn die Programmiersprache nicht sehr mächtig ist, ist die Konsequenz, daß der Benutzer zu stark weg vom Problem und zu stark in sprachspezifische Konstrukte zur Lösung von Detailproblemen geführt wird.

Höbersprachliche Hilfsmittel für die Parallelprogrammierung stellen Konstrukte zur Verfügung, die eine Formulierung parallel ablaufender Aktivitäten in einer mehr dem Problem angepaßten (symbolischen) Form ermöglichen. Dazu kann die Erstellung eines eigenen Compilers oder die Erweiterung eines bestehenden Compilers erforderlich werden.

Die in EGPA implementierten Makro-Datenflußmechanismen [Knei82] sind ein Beispiel. Sie führen von system- und sprachspezifischen Begriffen wie "Segment", "Prozeß" weg zu Begriffen wie "Funktion", "Argument", "Ergebnis". Parallel ablaufende Aktivitäten entstehen durch gleichzeitiges Anwenden von verschiedenen Funktionen auf ihre jeweils verfügbaren Argumente, wobei jede Funktionsanwendung nach endlicher Zeit ein Ergebnis liefert. Kommunikation zwischen Prozessoren erfolgt durch Konsumieren der Argumente bzw. Ergebnisse von Funktionen, die auf fremden Prozessoren ablaufen bzw. abgelaufen sind. (Argumente und Ergebnisse sind die einzige Form von Daten, auf die mehr als ein Prozeß zugreifen kann.)

Funktionen, Argumente und Ergebnisse werden für jeden Prozessor in Form eines symbolischen Ausdrucks (Datenflußkontrollausdruck) verwaltet, der nach festen Regeln verändert werden kann. Die Programmierung parallel verlaufender Aktivitäten geschieht durch Umformung der symbolischen Ausdrücke auf verschiedenen Prozessoren (z. B. als Einbringen einer Funktion mit ihren Argumenten in den Datenflußkontrollausdruck des fremden Prozessors, die von diesem konsumiert werden). Die Regeln zur Umformung symbolischer Ausdrücke stellen eine eigene Programmiersprache dar, die in die konventionelle Sprache eingebettet wird. Zur Umformung dieser Sprache in die Ebene der elementaren Hilfsmittel wird ein Vorübersetzer verwendet. Eine ausführliche Beschreibung wird in [Knei82] gegeben.

Die Verwendung einer solchen Schnittstelle erlaubt es viel stärker als auf der Ebene der elementaren Hilfsmittel, benutzerspezifische Informationen eines Parallelprogramms in die Ebene des Programmiersystems einzubringen. Insbesondere in Fehlerfällen bei Programmentwicklung und zur Laufzeit kann über ein solches System eine sehr viel detailliertere Analyse über den Kontext eines Fehlers an den Benutzer weitergegeben werden, z. B. welche Benutzerfunktionen auf welchen Prozessoren gerade aktiv waren, welche Argumente und Ergebnisse noch nicht konsumiert sind usw.

Auf der anderen Seite besteht jedoch die Gefahr, auf einer neuen Ebene Schwierigkeiten bei der Programmierung aufzuwerfen, die "systemspezifische" Lösungen verlangen. Auch Fragen der Effizienz müssen in diesem Rahmen angeschnitten werden (mögliche Verlangsamung der Programme, zusätzliche Anforderung an Speicher für Programme und Daten), ähnliche Probleme, wie sie beim Übergang von der Assemblerprogrammierung zur Programmierung in einer Höheren Programmiersprache zu beachten sind.

Auf einige dieser Fragen wird in den nachfolgenden Kapiteln eingegangen. In Kap. 6 wird eine der "E/A-orientierten" Methode zur Entwicklung von Parallelprogrammen angepaßte höbersprachliche Programmierungsumgebung skizziert.

5.2 Kommunikation und Synchronisation

Bei der Realisierung von Parallelprogrammen über die "E/A-orientierte" Implementierungsmethode wurde bislang die Frage offen gelassen, wie die Kommunikationsprozeduren als Mittel zur Übertragung von Information zwischen verschiedenen Prozessoren des Multiprozessorsystems zu implementieren sind.

Nachfolgend sollen für zwei Fragestellungen in diesem Bereich Lösungsmöglichkeiten vorgestellt und bewertet werden. Kap. 5.2.1 stellt Möglichkeiten vor, Puffer zur Aufnahme von zwischen Prozessoren fließender Information im gemeinsam zugreifbaren Speicher bzw. als Teil einer zwischen Prozessoren übermittelten

Botschaft zu realisieren. In Kap. 5.2.2 wird auf die eng damit in Zusammenhang stehende Frage der Synchronisation zur Sicherung eines korrekten Informationsflusses eingegangen. Im Kap. 5.1 skizzierte Programmierungsumgebungen dienen als Beispiele zum Anwendungsbereich der skizzierten Mechanismen. Im Kap. 4 dargestellte Anwendungsbeispiele werden zur Gewinnung "quantitativer" Aussagen bezgl. verschiedener Realisierungen herangezogen.

5.2.1 Möglichkeiten der Realisierung von Kommunikationspuffern

Eine naheliegende Möglichkeit zur Realisierung eines Speicherbereichs, aus dem eingetragene Informationseinheiten immer in der gleichen Reihenfolge wie bei der Eingabe gelesen werden,⁴⁶⁾ bildet ein F I F O - P u f f e r, der als Feld von Elementen des Typs der auszutauschenden Information oder auf der Basis von Bytes deklariert werden kann.

Neben der eigentlichen Felddefinition werden als Größen noch ein Zeiger benötigt, der anzeigt, wohin die nächste einzutragende Information geschrieben werden kann, und ein Zeiger auf die Feldposition, von der die nächste Information gelesen werden kann. Synchronisationsroutinen (vgl. Kap. 5.2.2) müssen dafür sorgen, daß beim Lesen Information im Puffer steht und beim Schreiben noch nicht gelesene Information nicht überschrieben wird. Die Zeiger sind als Umlaufzeiger realisiert, d. h. bei Erreichen des "obersten" Pufferelementes wird auf das erste Pufferelement weitergeschaltet.

46) Mit Hilfe solcher Operationen lassen sich auch Reihenfolgebbedingungen bei der Ausgabe von Ergebnissen für die parallelen Verarbeitungsstrategien für ordnungserhaltende Programmsysteme des Kap. 3.2 sichern. Bei Neuvergabe eines Auftrags wird die Kennung des betreffenden Arbeitsprozessors als Information in einen Auftragspuffer eingetragen und bei Auftragsende entsprechend wieder ausgelesen.

Bereich der Verweilzeiten bei STRING	Pufferelementen vom Typ BYTE
125.13 sec - 128.65 sec	132.61 sec - 135.95 sec

Abb. 5.1: Verweilzeiten für die Textformatierung mit verschiedenen Puffertypen
(Beispiel: Formatierung eines 47 Seiten langen Textes mit einer Zuordnung von Programmfunktionen zu Prozessoren)

Ob die Feldelemente der FIFO-Puffer in Form von Elementen beliebigen Typs oder als Bytes deklariert werden, hängt neben Überlegungen der allgemeinen Verwendbarkeit von Fragen des verfügbaren Platzes und Geschwindigkeitsüberlegungen ab. Abb. 5.1 zeigt, wie bei der Textformatierung mit der Verarbeitungsform "Zuordnung von Funktionen zu Prozessoren" die Art der Deklaration der Pufferelemente und die damit verbundene Form der Informationsspeicherung (bei Bytepuffern gepackte Speicherung von Textzeilen) die Verweilzeit beeinflusst. Die Verarbeitungsform mit Bytepuffern resultiert in einer etwa 6 % höheren Verweilzeit als die Verarbeitungsform mit Stringpuffern. Bei der Textformatierung auf der Basis der Aufteilung des Datenbestandes auf Prozessoren (Absätze) war eine Speicherung von Zwischenergebnissen in Form von Strings aus Platzgründen überhaupt nicht möglich.

Dies ist ein weiteres Beispiel dafür, wie scheinbare Nebensächlichkeiten signifikante Auswirkungen auf Programmlaufzeiten (auch schon von sequentiellen Programmen) haben können und damit die Fragwürdigkeit reiner Geschwindigkeitsvergleiche zwischen sequentiellen und parallelen Implementierungen offenbaren.

FIFO-Puffer haben als Elemente zur Aufnahme von zwischen Prozessoren zu übertragender Information neben ihrem Vorzug der einfachen Implementierbarkeit (vgl. hierzu auch Kap. 5.2.2 und Kap. 5.3) jedoch zwei Nachteile, die Grund für die Untersuchung der Anwendbarkeit der nachfolgend beschriebenen Mechanismen sind:

1. Für jeden zwischen Prozessoren fließenden Informationsstrom (innerhalb der Kommunikationsprozeduren durch verschiedene Kanäle realisiert) ist die Anlage eines eigenen Puffers erforderlich. Dies bringt bei mehr als einem Informationsstrom zwischen Prozessoren neben dem erhöhten Deklarationsaufwand die Problematik mit sich, wie verschiedene Puffer zu dimensionieren sind. Es kann sich als notwendig erweisen, für die Mindestgröße eines bestimmten Puffers einen bestimmten Wert vorzuschreiben und den Puffer während des Programmlaufs die meiste Zeit nicht zu benutzen bzw. leer stehen zu lassen und damit Speicherplatz zu verschwenden (Beispiel: Abbildungen und Fußnoten innerhalb des Textformatierungsprogramms).
2. Wenn nicht jede Informationseinheit eindeutig von anderen Informationseinheiten des gleichen Typs unterschieden werden kann (z. B. in Puffern gehaltene Absätze), muß entweder jede Informationseinheit in einem eigenen Puffer abgespeichert werden oder Trennmarkierungen zwischen verschiedenen Informationseinheiten des gleichen Typs eingefügt werden.

Die geschilderten Nachteile lassen sich mit Systemen vermeiden, die es gestatten, einen verfügbaren Speicherbereich flexibel den Anforderungen entsprechend aufzuteilen.

Ein Beispiel hierzu ist die Verwaltung des Speichers durch verkettete Listen. Abb. 5.2 zeigt, wie im Fall der Textformatierung Absätze in Form verketteter Listen gespeichert werden können. Eine Speicherung von Absätzen in Form verketteter Listen benötigt im Vergleich zur Speicherung von Absätzen in nicht verzeigerten Stringfeldern weniger Platz, da wegen der Verzeigerung unterschiedlich lange Zeilen innerhalb von Absätzen und unterschiedlich lange Absätze im Speicher an

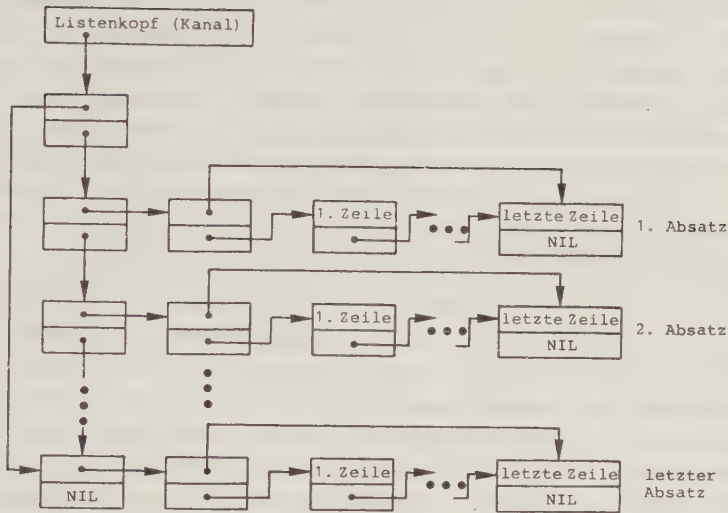


Abb. 5.2: Speicherung von Absätzen in Form verketteter Listen

beliebiger Stelle in kompakter Form (als Kette aufeinanderfolgender Bytes) angelegt werden können.

Die Vorzüge der Speicherung mit verketteten Listen sind unmittelbar aus Abb. 5.2 ersichtlich. Zum einen findet eine klare Trennung verschiedener Informationseinheiten statt. Zum anderen ist es möglich, den Speicher besser auszunutzen, indem im wesentlichen nur der Platz reserviert werden muß, der tatsächlich von Information belegt ist, d. h. es findet kaum Speicherverschnitt statt. (Der Platz, der zusätzlich zur Speicherung der Verkettungszeiger⁴⁷⁾ zu berücksichtigen ist, kann im Vergleich zu der gespeicherten Menge an Information vernachlässigt werden.) Auch eine Abbildung von E/A-Kanälen in diese Form kann einfach vollzogen werden; verschiedene Kanäle entsprechen verschiedenen Listenköpfen.

Hauptgrund für die Untersuchung dieser Abspeicherungsform war jedoch die bereits für die sequentiellen Programme zur Textverarbeitung (Editor, Textformatiersystem) vorhandene Software [Knei80], die sich einfach auf die Bedürfnisse der Parallelverarbeitung erweitern ließ. Es wurden lediglich Routinen eingefügt, die eine Manipulation des Listenspeichers (Einfügen neuer Elemente, Ausketten nicht mehr benötigter Elemente, Speicherbereinigung) zur Konsistenzerhaltung unter wechselseitigem Ausschluß sicherten, d. h. das Listenpaket wurde in Form eines Hoare'schen Monitors [Hoa74] implementiert.

Folgende Gründe waren jedoch dafür ausschlaggebend, daß dieses Speicherkonzept nicht weiter verfolgt wurde:

1. Es ist schwierig, die Aufnahmefähigkeit des Listenspeichers zu bewerten. Die Anzahl freier Listenelemente allein ist kein Kriterium. Es ist darüberhinaus die Art der Stückelung des Speichers in freie und belegte Bereiche zu berücksichtigen, die maßgeblich mitbestimmt, wann der nächste Speicherbereinigungsvorgang zu erfolgen hat. Die effiziente Implementierung speicherorientierter Heuristiken verlangt jedoch eine "schnelle" Ermittlung dieser Größen.
2. Im Fehlerfall (Ablegen von Information aufgrund von Speicherengpässen nicht möglich) ist es aufwendig, die Ursachen darzustellen und für einen "ordnungsgemäßen" Programmabbruch zu sorgen (vgl. Kap. 5.3.5).
3. Die Manipulation des Listenspeichers unter wechselseitigem Ausschluß ist häufig nicht unbedingt notwendig, was sich jedoch vorher nicht einfach erkennen läßt. Vor allem dann, wenn der Ablauf der Routine zur Freispeichersammlung erforder-

- 47) Eine zum Konzept der verketteten Listen äquivalente Form der Speicherverwaltung, bei der auf Verkettungszeiger verzichtet werden kann, läßt sich durch einen Assoziativspeicher realisieren. In diesem Fall wird ein zu bearbeitendes Element nicht über einen Zeiger, sondern über ein bestimmtes Suchmuster (assoziativ) aufgefunden.

derlich wird, wird ein anderer Prozeß, der den Inhalt des Listenspeichers verändern möchte, eine beträchtliche Zeit aufgehalten.⁴⁸⁾

Eine Speicherungsform, die Verzögerungen von Prozessen aufgrund exklusiv zu durchlaufender Abschnitte weitgehend vermeidet und trotzdem eine den Anforderungen entsprechende Stückelung des verfügbaren Kommunikationsspeichers erlaubt, ist die in Form D F C⁴⁹⁾ - D a t e i e n.

Dieses Konzept zur Speicherverwaltung, welches Bestandteil der in Kap. 5.1.3 skizzierten Makro-Datenflußmechanismen [Knei82] ist, sieht vor, zwischen Prozessoren zu übertragende Informationseinheiten in eigens reservierten Datenbereichen, den DFC-Dateien, zu speichern. Es existieren damit neben den Aufrufen zum Beschreiben und Lesen einer DFC-Datei Dienste zum Anlegen eines solchen Bereichs bestimmter Größe und bestimmten Typs bzw. zum Löschen. Verschiedene DFC-Dateien werden von den Prozessoren über Referenzen⁵⁰⁾ erreicht, die beim Anlegen der Datei an die betreffenden Prozessoren gemeldet werden (vgl. Abb. 5.3). Exklusive Zugriffe auf den gemeinsamen Speicher sind nur dann notwendig, wenn eine neue Datei angelegt wird oder eine nicht mehr benötigte Datei gelöscht wird (nicht jedoch beim Lesen und Beschreiben von DFC-Dateien). Die Aktualisierung der den verfügbaren Speicher beschreibenden Größen (Zahl, Größe und Lage freier und belegter Bereiche) erfolgt in diesem Fall mit Ausführung der entsprechenden Operation, d. h. es entfällt eine spezielle Freispeichersammlung und eine Auskunft über den Be-

48) Vergleichsmessungen bei der Textformatierung zwischen Parallelprogrammen mit einer Prozessorkommunikation über FIFO-Puffer zu Parallelprogrammen mit einer Prozessorkommunikation über Listenspeicher haben ergeben, daß letztere eine bis zu 25 % (!) höhere Verweilzeit hatten.

49) Data Flow Control.

50) Bei einer Realisierung des Kommunikationsspeichers als assoziativer Speicher lassen sich verschiedene DFC-Dateien über charakteristische Suchmuster erreichen.

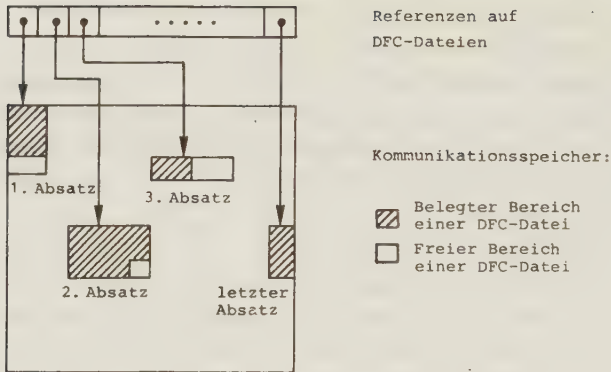


Abb. 5.3: Abspeicherung von Absätzen in Form von DFC-Dateien

gungszustand des Kommunikationsspeichers ist immer aktuell und "einfach" zu erhalten.

Das Konzept der DFC-Dateien hat jedoch auch einige Nachteile:

1. Bei Anlage jeder neuen DFC-Datei muß eine Abschätzung über den Platzbedarf der aufzunehmenden Informationseinheit durchgeführt werden. Bei Datenblöcken, bei denen aufgrund von zuvor abgelaufenen Programmteilen keine solche Abschätzung möglich ist (z. B. nicht formatierte Absätze, bei denen die Länge von vornherein nicht feststeht), muß immer so viel Platz bereitgestellt werden, daß die größte zu erwartende Informationseinheit aufgenommen werden kann. Wenn viele Informationseinheiten zwischengespeichert werden (wovon die parallelen Verarbeitungsstrategien für ordnungserhaltende Programmsysteme des Kap. 3 ausgehen), kann sich ein starker Speicherverschnitt ergeben, aufgrund dessen ein Anlegen einer neuen DFC-Datei nicht möglich ist, obwohl in der Summe genug freier Speicher zur Aufnahme der darin abzulegenden Information bereitstehen kann.

2. Wenn eine neue DFC-Datei nicht angelegt werden kann, muß dies nicht auf einen Speicherengpaß zurückzuführen sein, sondern kann daran liegen, daß die Zahl der zu referenzierenden Informationseinheiten zu groß wird.
3. Es können sich wegen der nur exklusiv ausführbaren Operationen zur Aktualisierung der einen DFC-Speicher beschreibenden Größen zusätzliche Verzögerungen im Ablauf der Prozesse ergeben. Dies wirkt sich vor allem dann aus, wenn im Vergleich zur Zahl der prozeßlokal ausführbaren Operationen sehr häufig zwischen Prozessen durch Anlegen und Löschen von DFC-Dateien kommuniziert wird.

Die im letzten Punkt angesprochenen Verzögerungen können im Extremfall Programmlaufzeiten in der gleichen Größenordnung beeinflussen wie die Manipulation des Listenspeichers unter wechselseitigem Ausschluß. Bei Verwendung von FIFO-Puffern fester Größe ist es jedoch möglich (vgl. Kap. 5.2.2), derartige durch "Systemeinflüsse" (die Speicherverwaltung) bedingte Blockaden zu vermeiden.

Vor allem die Tatsache, daß Probleme der Synchronisation und Behandlung möglicher Systemverklemmungen (vgl. Kap. 5.3) mit FIFO-Puffern am besten bewältigt werden können und diese den geringsten Aufwand bei der Beachtung systemspezifischer Einflüsse erfordern, hat den Ausschlag dafür gegeben, daß in den durchgeführten Implementierungen FIFO-Puffer als Speicher von zwischen Prozessoren zu übertragender Information bevorzugt wurden.

Es sollte jedoch darauf hingewiesen werden, daß die Probleme im Zusammenhang mit den anderen Speicherkonzepten nicht primär auf strukturell bedingte Eigenschaften zurückzuführen sind. Die Schwierigkeiten bei der Verwendung des Listenpakets und der DFC-Dateien entstehen, weil die Kapazität des Kommunikationsspeichers eingeschränkt ist und Zeiten für die exklusiv zu durchlaufenden Operationen zur Manipulation der den Kommunikationsspeicher beschreibenden Größen im Vergleich zur Rechenzeit nicht vernachlässigt werden können.

Das Speicherverwaltungskonzept der DFC-Dateien wird gegenüber dem Konzept der FIFO-Puffer sicher uneingeschränkt vorzuziehen sein ("größere Flexibilität bei der Speicherung verschiedener Informationsströme", "geringerer Implementierungsaufwand auf Benut-zerseite"), wenn ein "schneller", "ausreichend" dimensionierter Assoziativspeicher als Kommunikationsspeicher zur Verfügung steht [Knei82].

In diesem Sinne mögen diese Betrachtungen als ein Beispiel für Schlußfolgerungen angesehen werden, die sich aus der Einschränkung verfügbarer Ressourcen in Multiprozessorsystemen ergeben und zu möglichen Konsequenzen, die sich aus der Aufhebung dieser Einschränkungen ergeben können.

5.2.2 Zur Implementierung von Synchronisationsmechanismen

Innerhalb der letzten Kapitel wurden Fragen der Synchronisation zwischen parallel ablaufenden Prozessen in einer informellen Art angesprochen ("die Synchronisation hat dafür zu sorgen, daß in Puffern niedergelegte Information nicht überschrieben wird" u. ä.). Hier soll die Rolle der Synchronisation innerhalb der in Kap. 2 dargestellten Implementierungsmethode in einer etwas präziseren Weise dargestellt werden und in diesem Zusammenhang auf einige aus der Literatur bekannter Konzepte eingegangen werden.

Fragen der Synchronisation zwischen parallel ablaufenden Prozessen innerhalb eines Parallelprogramms treten dann auf, wenn ein Programmfortschritt eines Prozesses vom Verlauf eines anderen Prozesses abhängig ist. Die Synchronisation hat dann dafür zu sorgen, daß die davon betroffene Operation erst dann ausgeführt wird, wenn gewisse von einem anderen Prozeß beeinflusste Bedingungen erfüllt sind. Verschiedene Mechanismen zur Synchronisation unterscheiden sich hauptsächlich darin, wie solche zu einem Wartezustand führenden Bedingungen formuliert sind und auf der Seite der Implementierung, wie Stoppen laufender und Wiederaanlaufen wartender Prozesse realisiert werden.

Wenn Parallelprogramme über die in Kap. 2 beschriebene Methode entwickelt werden, treten Stellen, an denen Prozesse durch andere Prozesse beeinflußt werden, nur innerhalb von Kommunikationsprozeduren auf. Parallele Verarbeitungsstrategien des Kap. 3 berücksichtigen Zustände anderer Prozesse außer in Kommunikationsprozeduren nur noch innerhalb des Moduls, in dem die Strategie formuliert wird. Wenn Synchronisation innerhalb von Kommunikationsprozeduren in der Form der Abb. 5.4 formuliert wird (die mit (*) bezeichneten Synchronisationsoperationen als Prozeduren), kann die Synchronisation als eigenständiger abstrakter Datentyp innerhalb eines separaten Moduls formuliert werden. Die Trennung des Synchronisationsteils von den übrigen Funktionen ist auch Grundlage eines in [Ker82] beschriebenen Verfahrens zur Implementierung asynchroner Systeme. Damit ergibt sich aus der Modulhierarchie des sequentiellen Programms (vgl. Abb. 2.18) die in Abb. 5.5 dargestellte Aufrufhierarchie der Module E/A-orientierter paralleler Implementierungen.

In dieser verdeutlicht sich, abgesehen von der Ausgestaltung des parallelen Strategiemoduls, der einzige wesentliche Unterschied zwischen E/A-orientierten sequentiellen und E/A-orientierten parallelen Implementierungen im Zustandsbegriff, wie er innerhalb des Kommunikationsmoduls verwendet und nach Ausführung einer Kommunikationsprozedur zurückgemeldet wird. Während in sequentiellen Programmen vor Ausführung der Kommunikationsprozedur nur die beiden Ergebnisse "Operation korrekt ausführbar" oder "Operation aufgrund spezifizierbarer Ereignisse nicht ausführbar" möglich sind, existiert in Parallelprogrammen an dieser Stelle noch zusätzlich der Zustand "Operation momentan nicht ausführbar", der nach Abschluß der Warte-Routine (durch eine auf einem anderen Prozeß ausgeführte Signalisierungsroutine) in einen der beiden anderen Zustände übergeführt worden ist. (Auf die Behandlung von Fehlerzuständen und daraus möglicherweise resultierenden Systemverklemmungen wird in Kap. 5.3 und 5.4 eingegangen.)

MODULE PARALLELEA

...

GLOBAL SUBR READINFO = (INT KANAL, <INFO bel. Typs>) INT
BEGIN

```
    INT ZUSTAND;  
    ZUSTAND := "Warte, bis INFO verfügbar  
                oder Fehlerzustand vorliegt";    (*)  
    IF ZUSTAND = "INFO verfügbar"  
    THEN "INFO lesen";  
        "Melde Ausführung des Lesevorgangs";    (*)  
        RETURN("Korrektter Verlauf")  
    ELSE RETURN(ZUSTAND)  
    FI
```

END;

GLOBAL SUBR WRITEINFO = (INT KANAL, <INFO bel. Typs>) INT
BEGIN

```
    INT ZUSTAND;  
    ZUSTAND := "Warte, bis INFO abgelegt werden kann  
                oder Fehlerzustand vorliegt";    (*)  
    IF ZUSTAND = "INFO ablegbar"  
    THEN "INFO schreiben";  
        "Melde Ausführung des Schreibvorgangs";    (*)  
        RETURN("Korrektter Verlauf")  
    ELSE RETURN(ZUSTAND)  
    FI
```

END;

...

MODEND

Abb. 5.4: Kommunikationsprozeduren in Parallelpro-
grammen (vgl. [Ker82])

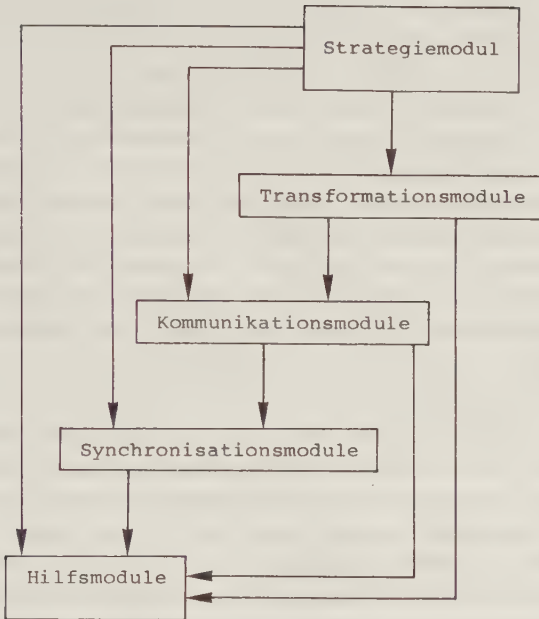


Abb. 5.5: Aufrufhierarchie der Module E/A-orientierter paralleler Implementierungen

Die Gestaltung von Warte- und Signalisierungsroutinen ist von der Art des verwendeten Synchronisationsmechanismus abhängig. Der modulare Aufbau E/A-orientierter Parallel-Implementierungen, wie er in Abb. 5.5 dargestellt ist, erlaubt das Testen verschiedener Synchronisationsmechanismen von der speziellen Anwendung losgelöst durch Austausch der zugehörigen Synchronisationsmodule.

Die Gestaltung der Synchronisation wird maßgeblich davon beeinflusst, welche Hilfen das Betriebssystem zur Prozeßsteuerung zur Verfügung stellt. In diesem Kontext läßt sich eine Einteilung in zwei Entwicklungslinien vornehmen, botschaftsorientierte Systeme und prozedurorientierte Systeme.

Kennzeichen botschaftsorientierter Systeme ist, daß die Kommunikation zwischen verschiedenen Prozessen nur über Botschaften erfolgt. Datenstrukturen, die von mehr als einem Prozeß manipuliert werden, müssen von einem zum anderen Prozeß als Bestandteil einer Botschaft übergeben werden. Kennzeichen prozedurorientierter Systeme ist, daß verschiedene Prozesse auf einen gemeinsam benutzbaren Speicherbereich zugreifen können. Informationsübergabe und Prozeßsteuerung erfolgt durch Manipulation von Daten im zwischen den Prozessen gemeinsam benutzten Speicherbereich.

Ein Vergleich zwischen den beiden Organisationsformen findet sich in [LaNe79]. Die in Kap. 5.1.1 beschriebenen elementaren Hilfsmittel zur Erstellung von Parallelprogrammen im (hier verwendeten) EGPA-System stehen stellvertretend für eine prozedurorientierte Programmierumgebung, die in Kap. 5.1.2 beschriebenen elementaren Hilfsmittel in einem UNIX-System stehen stellvertretend für eine botschaftsorientierte Programmierumgebung. Die höhersprachlichen Programmierhilfsmittel auf der Ebene des Makro-Datenflusses (vgl. Kap. 5.1.3) stellen eine Zwischenform dar. Information, die zwischen verschiedenen Prozessen ausgetauscht wird, wird zwar in der Art einer Botschaft von einem Prozeß an den anderen weitergereicht. Weitergegeben wird jedoch nicht der Inhalt der Information, sondern eine Referenz auf die Stelle im gemeinsam benutzten Speicher, an der die entsprechende Information abgelegt ist, sodaß Prozeßsteuerungen über Manipulation von Daten im gemeinsam benutzten Speicher möglich sind.

Vorzüge botschaftsorientierter Systeme (z. B. [Fel79], [Han78], [Hoa78], [Iss83], [Lis79]) sind, daß Prozeßkommunikation und Prozeßsynchronisation (die nur über den Botschaftsmechanismus ablaufen) leichter auf einer Systemebene implementiert und kontrolliert werden können. Prozesse haben aus Benutzersicht einen voll privaten Adreßraum, d. h. es gibt aus Benutzersicht keine gemeinsam benutzten Code- oder Datensegmente und damit auch keine Notwendigkeit, Speicherbereiche in bestimmten Situationen im Programmablauf als exklusiv zugreifbar zu kennzeichnen. Nachteile botschaftsorientierter Systeme gegenüber prozedurorientierten Systemen liegen darin, daß es aufwendiger ist, Information zwischen Prozessen auszutauschen und einen (für eine Analyse von Programmfehlern notwendigen) globalen Zustand der Prozesse im Parallelprogramm festzustellen.

Ein Weg, diese Nachteile botschaftsorientierter Systeme zu vermeiden, wird in [JaSt77] im Konzept der Managerprozesse bzw. in [Lis82] über Guardians aufgezeigt. In Botschaften niedergelegte Information wird von speziell zur Synchronisation eingerichteten Prozessen verwaltet. Über diese Prozesse wird die eigentliche Synchronisation abgewickelt. Durch Analyse des von diesen Synchronisationsprozessen verwalteten Botschafts- und Datenbestands kann auch relativ einfach ein Abbild vom Zustand des ein Parallelprogramm realisierenden Prozeßsystems gewonnen werden.

Untersuchungen zur Gestaltung von Mechanismen zur Synchronisation von Prozessen in prozedurorientierten Systemen werden seit etwa fünfzehn Jahren durchgeführt. In [Bath82] findet sich eine Zusammenstellung und "Leistungs"-Klassifikation von über vierzig innerhalb der letzten Jahre vorgeschlagenen Synchronisationsmechanismen. "Schwache" Mechanismen im Sinn der Klassifikation von [Bath82] wie Semaphore [Dij68] haben dabei gegenüber "starken" Mechanismen wie "Bedingten Kritischen Abschnitten" [Han73] neben den Schwierigkeiten, "komfortabel" Wartebedingungen zu formulieren, vor allem den Nachteil, daß häufig Prozesse von anderen Prozessen an der weiteren Programmausführung aufgrund

exklusiv zu durchlaufender Sequenz gehindert werden, obwohl dies aufgrund der Zustände der beteiligten Prozesse nicht notwendigerweise erforderlich wäre [Han72].

Für die Implementierung E/A-orientierter Parallelprogramme, bei denen Information zwischen verschiedenen Prozessen über FIFO-Puffer ausgetauscht wird, bietet sich eine Synchronisation über Ereigniszähler [Ger77], [ReKa79] an.

Für jeden Puffer werden zwei Zähler definiert. In einem Zähler wird gezählt, wieviel Information (z. B. in Bytes) in den betreffenden Puffer eingetragen wurde, in einem anderen Zähler, wieviel Information vom betreffenden Puffer gelesen wurde. Wenn mit WRITEANZ die Anzahl der geschriebenen Einheiten, mit READANZ die Anzahl der gelesenen Einheiten und mit GROESSE die Kapazität des entsprechenden Puffers bezeichnet wird, müssen die Synchronisationsroutinen nur sicherstellen, daß gilt:

$$0 \leq \text{WRITEANZ} - \text{READANZ} \leq \text{GROESSE}$$

Wie in [ReKa79] ausgeführt, hat dieser Synchronisationsmechanismus in diesem speziellen Fall des Ein-Erzeuger-Ein-Verbraucher-Problems gegenüber einer Lösung mit Semaphoren den Vorteil, daß auf die Notwendigkeit einer Blockade von Prozessen durch Ausführen kritischer Abschnitte verzichtet werden kann. Damit wird der Anforderung an "gute" Synchronisationsmechanismen Rechnung getragen, die nur dann ein paralleles Arbeiten von Prozessen verhindern, wenn dies aus Benutzersicht "gewollt" ist.

In Kap. 5.3 wird gezeigt, daß Ereigniszähler auch grundsätzlich für die Verifikation paralleler Programme vor allem im Hinblick auf die Untersuchung von Systemverklemmungen nützlich sind.

Für die eigentliche Implementierung der Synchronisation bieten sich zwei Verfahren an, eine Realisierung über zyklisches Auswerten der Wartebedingung (Aktives Warten) oder über Verwendung der vom Betriebssystem bereitgestellten Software-Interruptmechanismen.

Aktives Warten bietet für die Implementierung den Vorteil, daß Routinen zur Beseitigung möglicher Wartezustände nicht danach "schauen" müssen, ob evtl. ein blockierter Prozeß zu befreien ist. Nachteil ist jedoch die Belastung des Kommunikationsspeichers, die einen anderen auf diesem Speicherbereich arbeitenden Prozeß behindern kann. (Die Belastung der CPU durch Leerarbeit, die in Timesharing-Betriebssystemen im allgemeinen nicht toleriert wird, spielt im Rahmen der untersuchten Problemstellungen eine eher untergeordnete Rolle.) Die Belastung des Kommunikationsspeichers durch Leerarbeit wird bei Verwendung von Interruptmechanismen vermieden. Dafür kann es unter Umständen recht aufwendig sein, einmal durch Interrupt blockierte Prozesse nach Aufheben der Wartebedingung wieder zu befreien.

Für die Implementierung der Synchronisation auf dem EGPA-System wurde folgender Kompromiß durchgeführt. Bei der Synchronisation zwischen Kontrollprozessor (B-Prozessor) und Arbeitsprozessoren (A-Prozessoren) werden nur dann Interrupts abgesetzt, wenn wegen einer momentan nicht erfüllten Bedingung gewartet werden muß.⁵¹⁾ Für den Fall, daß Arbeitsprozessoren aufeinander warten müssen, wurde von "Systemseite" eine Unterstützung von benutzerimplementiertem Aktivem Warten bereitgestellt (vgl. Kap. 5.3.4).

Gründe hierfür liegen zum einen in den Zeitverlusten⁵²⁾, die durch die Verwendung des Interruptmechanismus speziell im Verkehr der Arbeitsprozessoren untereinander entstehen (vgl. Abb. 5.6). Für den Verkehr zwischen Kontrollprozessor und Arbeitsprozessoren gilt dies nicht in analoger Weise. Zum einen ist hier der Zeitverlust geringer. Zum anderen erlaubt es dieses Verfahren besser als das Aktive Warten, Unterbrechungen im Programmablauf

51) Dies entspricht einer Realisierung der Synchronisation über binäre Semaphore (vgl. hierzu auch [FHHJ83]).

52) Daß Synchronisationsvorgänge zwischen zwei Prozessoren über Polling (Aktives Warten) in der Regel schneller ablaufen als Synchronisationsvorgänge über Interrupts, wird auch durch Untersuchungen am DIRMU-Multiprozessorssystem [Hai84] bestätigt.

Bereich der Verweilzeiten bei Synchronisation über	
Aktives Warten	Interrupts (bin. Semaphore)
125.13 sec - 128.65 sec	131.04 sec - 134.40 sec

Abb. 5.6: Verweilzeiten für die Textformatierung mit verschiedenen Implementierungen der Synchronisation (Beispiel: Formatierung eines 47 Seiten langen Textes mit einer Zuordnung von Programmfunktionen zu Prozessoren)

auf dem Kontrollprozessor vorzunehmen. Dies ist von Bedeutung, da der Kontrollprozessor im EGPA-System auch von anderen Benutzern gleichzeitig im Timesharing benutzt wird.

Die Zeitverluste bei Verwendung des Interruptmechanismus sind auch dann sehr gering, wenn ein Wartevorgang sehr selten auftritt (vgl. Abb. 5.7). Sie lassen sich auch durch "geschicktes" Programmieren minimieren, nämlich erst dann auf ein Resultat zu warten, wenn dieses unbedingt benötigt wird bzw. einen Arbeitsprozessor zum frühest möglichen Zeitpunkt mit neuer Arbeit zu versorgen, um echte Wartephase weitgehend einzusparen (Prinzipien "Eingaben so spät wie möglich", "Ausgaben so früh wie möglich").⁵³⁾

Die ermittelten Ergebnisse legen den Schluß nahe, daß für die Implementierung der Synchronisation in den Parallelprogrammen des untersuchten Typs auf die Verwendung von Interruptmechanismen verzichtet werden sollte, wenn eine Implementierung der Synchronisation über Aktives Warten möglich ist. Für dedizierte

53) Bei einigen Anwendungen konnte eine Verkürzung der Laufzeit durch Hinzufügen von Programmcode erreicht werden. Dies ist daraus zu erklären, daß ein geringfügig längeres Rechnen das Ansprechen von Interrupts überflüssig machte. Derartige Effekte sind jedoch praktisch nicht vorhersehbar.

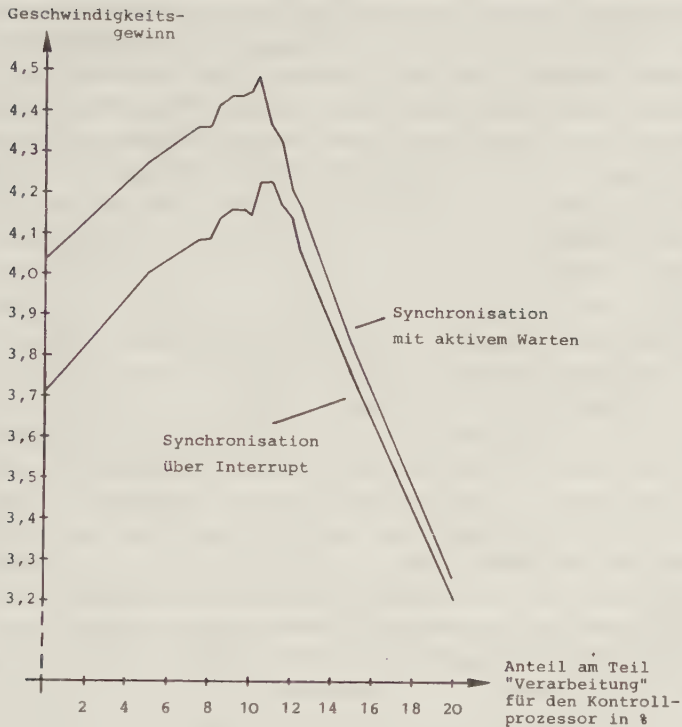


Abb. 5.7: Geschwindigkeitsgewinne für parallele Implementierungen der Netzwerkanalyse mit verschiedenen Synchronisationsmechanismen (Je mehr der Kontrollprozessor zu tun hat, desto weniger muß dieser auf Arbeitsprozessoren warten; desto geringer sind die Unterschiede zwischen verschiedenen Synchronisationsmechanismen.)

Systeme, in denen eine korrekte Zusammenarbeit der miteinander kooperierenden parallelen Prozesse gesichert ist, trifft dies sicher zu. Für Systeme jedoch, bei denen mit dem Auftreten nicht "erwarteter" Fehler gerechnet werden muß, kann es sich zur Aufrechterhaltung eines Multiprogramming-Betriebes als notwendig erweisen, fehlerhaft ablaufende Prozesse zwangsweise (durch Timeout oder Eingriff des Benutzers) zu beenden. In solchen Fällen (von denen letztlich auch im Rahmen dieser Untersuchungen zunächst ausgegangen werden muß) kann auf Interrupts nicht verzichtet werden [Hof84].

Abgesehen davon kann auf Interrupt-Mechanismen schon deshalb nicht verzichtet werden, da in Betriebssystemen üblicherweise Problemstellungen auftreten, in denen eine Synchronisation zwischen Prozessen über Interrupts schneller abläuft als eine Synchronisation über Aktives Warten. Man stelle sich etwa einen Betriebssystemprozeß zur Verwaltung nur exklusiv verwendbarer Geräte vor, welche von "beliebig" vielen Prozessen potentiell angefordert werden können aber kaum (im Vergleich zur sonstigen Laufzeit der Prozesse) benutzt werden. In einem solchen Fall ist zu erwarten, daß ein Benutzerprozeß bei Reaktion des Verwalterprozesses auf einen Interrupt schneller bedient wird als wenn der Verwalterprozeß (in möglicherweise 99 % der Fälle vergeblich) in einer aktiven Warteschleife über alle Benutzerprozesse potentielle Zugriffswünsche zu koordinieren hätte.

Die ausschließliche Verwendung einer parallelen Programmierungsumgebung zur Formulierung von Synchronisation in Parallelprogrammen erlaubt es, derartige Fragestellungen auf einer "Systemebene" (unabhängig vom jeweiligen Benutzerprogramm) abzuhandeln.

Abschließend sei noch eine Realisierung der Synchronisation durch Verzögertes Aktives Warten erwähnt. Diese wird in der Form des Aktiven Wartens gestaltet, wobei zwischen zwei Abfragen der Wartebedingung eine Verzögerung durch lokale (aber im übrigen sinnlose) Operationen vorgenommen wird.

Die zugrundeliegende Idee ist, daß eine weitere Programmausführung in Abhängigkeit einer Wartebedingung entweder sofort möglich ist oder eine signifikant lange Zeit gewartet werden muß. Ändern sich Wartebedingungen jedoch relativ schnell, kann immer noch erwartet werden, daß durch eine Verlängerung der Wartephase in einem Synchronisationsschritt ein Warten in weiteren Synchronisationsschritten gänzlich überflüssig wird.

Abb. 5.8 zeigt für das Beispiel "Textformatierung", daß es in

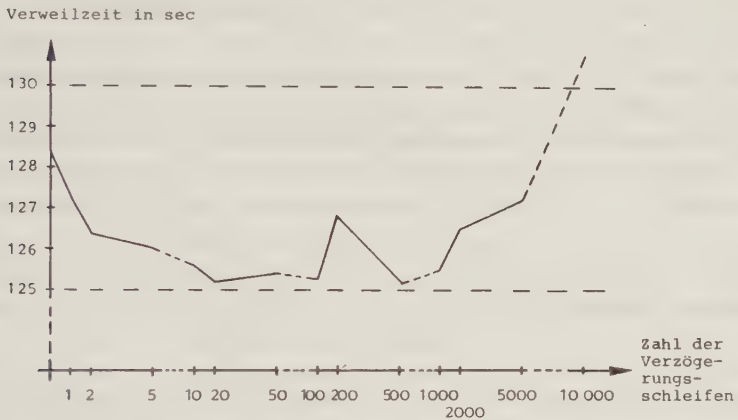


Abb. 5.8: Verweilzeiten bei der Textformatierung mit verschieden großen Verzögerungen zwischen zwei Auswertungen der Warte-Bedingung (Beispiel: Formatierung eines 47 Seiten langen Textes mit einer Zuordnung von Programmfunktionen zu Prozessoren)

einem weiten Bereich völlig belanglos ist, welche Zeit zwischen zwei Auswertungen einer Wartebedingung verbraucht wird. (In diesem Fall wurde in einer Laufschleife n -mal eine Gleitpunktoperation ausgeführt.) Die verzögerte Form des Aktiven Wartens erlaubt eine Reduzierung der Belastung des zwischen verschiedenen Prozessen gemeinsam benutzten Speichers, bietet auf

der anderen Seite jedoch alle Vorzüge, die das reine Aktive Warten bietet.

Mit der Implementierung von Mechanismen, die ein Warten von Prozessoren realisieren, besteht immer die Gefahr, daß kein Prozessor mehr in seinem Programm fortfahren kann, es also zu einer Systemverklemmung kommt. Wie solche Fälle erkannt und behandelt werden können, wird im Kap. 5.3 näher untersucht.

5.3 Erkennung und Beseitigung von Systemverklemmungen

Die Fragen der Behandlung von Systemverklemmungen in Parallelprogrammen berühren den Aspekt der Korrektheit. Es ist zwar möglich, den korrekten Ablauf einzelner Prozesse eines Prozeßsystems zu verifizieren. Damit ist jedoch nichts darüber ausgesagt, ob deren Zusammenspiel zum korrekten Gesamtergebnis führt.

Gerade in ordnungserhaltenden Programmsystemen, bei denen innerhalb eines Programms eine variabel große Menge von Daten zwischen Prozessoren transportiert wird, die Überlegungen bezgl. der Dimensionierung der die Daten aufnehmenden Puffer erfordern, kann nicht von vornherein ausgeschlossen werden, daß Pufferüberläufe drohen. In diesem Fall können zur Vermeidung von Informationsverlusten davon betroffene Prozesse nicht mehr in ihrem Programm fortfahren und bringen dann in der Folge die gesamten übrigen Prozesse des Systems (die vom korrekten Arbeiten der blockierten Prozesse abhängig sind) in einen nicht mehr auflösbaren Wartezustand.

Dies ist jedoch nur ein Spezialfall einer in Parallelprogrammen möglichen Systemverklemmung. Ganz allgemein besteht in einem Parallelprogramm dann eine Systemverklemmung, wenn jeder der das Programm steuernden Prozesse auf eine nicht erfüllbare Bedingung wartet. Systemverklemmungen können dabei auch auf Programmierfehler zurückzuführen sein, die den Ausfall eines Prozesses zur Folge haben.

Innerhalb von Betriebssystemen wurden Ursachen von Verklemmungen und ihrer Behandlung weitgehend untersucht. Eine Zusammenfassung von Resultaten findet sich in [CoES71] und [Hol72]. Diese umfassen jedoch nur Teilaspekte und können keine Lösung für das allgemeine Problem der Behandlung von Systemverklemmungen in Parallelprogrammen sein. Mit den nachfolgenden Ausführungen soll eine Möglichkeit aufgezeigt werden, wie innerhalb eines allgemeinen Kalküls zur Parallelprogrammierung [Hoa72] beschriebene Situationen einer Systemverklemmung [OwiG76] mit Hilfsmitteln einer Programmierumgebung behandelt werden können.

Ausgehend von einer Abgrenzung der Problematik der Behandlung von Systemverklemmungen in Betriebssystemen zu der in Parallelprogrammen werden verschiedene Algorithmen zur Erkennung von Verklemmungszuständen in Parallelprogrammen entwickelt und abschließend kurz auf Fragen der Diagnose und Beseitigung von Verklemmungszuständen eingegangen.

5.3.1 Situationsbeschreibung, Grundlagen zu Systemverklemmungen in Betriebssystemen und Parallelprogrammen

Untersuchungen zur Behandlung von Verklemmungszuständen in Betriebssystemen beruhen auf dem folgenden Modell.

Eine Menge parallel ablaufender Prozesse bewirbt sich während der Programmausführung um Betriebsmittel bestimmter Klassen wie Speicher, E/A-Geräte, Semaphore, Botschaften usw. Betriebsmittel werden wenn möglich zugeteilt und vom entsprechenden Prozeß nach endlicher Zeit wieder freigegeben. Zur formalen Beschreibung der Situation einer Systemverklemmung (Deadlock) werden an Daten benötigt:

BESITZ_{i,k}.....der augenblickliche Besitz des Prozesses P_i an Betriebsmitteln der Klasse B_k

FORDERUNG_{i,k}...die augenblickliche (über den Besitz hinausgehende) Forderung des Prozesses P_i an Betriebsmitteln der Klasse B_k , ohne daß irgendwelche Betriebsmittel freigegeben werden

VORHANDEN_k.....die Zahl der vorhandenen Betriebsmittel der Klasse B_k

Unter der Bedingung, daß zu keinem Zeitpunkt ein Prozeß von einer Betriebsmittelklasse mehr Exemplare verlangt als von dieser Klasse vorhanden sind, gilt in diesem Modell folgende Bedingung für ein in einer Verklemmung befindliches System paralleler Prozesse (vgl. [KNU75]).

Def. 5.1: Eine Menge $D=\{P_1, P_2, \dots, P_m\}$ von Prozessen befindet sich im Deadlock, wenn für alle $P_j \in D$ ein k_j existiert, sodaß gilt:

$$FORDERUNG_{j,k_j} > VORHANDEN_{k_j} - \sum_{i=1}^m BESITZ_{i,k_j}$$

Ein Verklemmungszustand bezgl. einer Menge parallel ablaufender Prozesse innerhalb eines Betriebssystems besteht also dann, wenn jeder Prozeß dieser Menge mindestens ein nicht zuteilbares Betriebsmittel anfordert, welches ein anderer Prozeß dieser Menge besitzt und selbst kein Betriebsmittel freigibt.

Die Folge ist, daß jeder Prozeß der "Deadlockmenge" auf die Freigabe von Betriebsmitteln durch andere Prozesse der "Deadlockmenge" wartet und sich dadurch eine geschlossene Kette aufeinander wartender Prozesse ergibt.

Wie in [KNU75] ausgeführt ist, verlangt diese Definition eines Deadlocks nicht, daß alle Prozesse des Prozeßsystems in D enthalten sind (partieller Deadlock). Sie erlaubt daher, Prozesse, die nicht im Deadlock sind, noch weiter auszuführen. Aber auch wenn alle diese Prozesse ihre Ausführung beenden und die von ihnen belegten Betriebsmittel freigeben, wird es nie eine Situation geben, so daß einer der Prozesse $P_j \in D$ eine weitere Aktion ausführen kann.

Von dieser Definition des Deadlocks ausgehend werden die folgenden Fragekomplexe untersucht:

1. Deadlockvermeidung:

Welche Bedingungen müssen bezgl. der Reihenfolge der Anforderungen der am System beteiligten Prozesse bestehen, damit im System kein Deadlock eintreten kann?

Nach welcher Strategie müssen Anforderungen der am System beteiligten Prozesse erfüllt werden, damit im System kein Deadlock eintreten kann?

2. Deadlockerkennung, Diagnose, Recovery:

Wenn ein Deadlock im System weder verhindert noch vermieden werden kann: Welche Möglichkeiten bestehen dann, einen evtl. aufgetretenen Deadlock zu entdecken, zu analysieren und zu beseitigen?

Da die Untersuchung der ersten beiden Fragestellungen von a priori bekannten Voraussetzungen ausgehen muß wie z. B. "Maximalbedarf der einzelnen Prozesse an Betriebsmitteln jeder Klasse bekannt", die in dieser Form innerhalb der untersuchten Parallelprogramme so nicht vorgegeben sein sollen, kann es in diesem Rahmen nur darum gehen, Methoden zur Entdeckung, Diagnose und Recovery von Deadlocks zu untersuchen.

Die nachfolgend entwickelten Algorithmen zur Erkennung von Systemverklebungen in Parallelprogrammen gehen von zwei Vorgaben aus, die für die Analyse auf mögliche Deadlocks innerhalb von Betriebssystemen in dieser Form im allgemeinen nicht zutreffen bzw. bedeutungslos sind.

1. Es wird nur eine Untersuchung totaler Systemverklebungen, durchgeführt, d. h. von Systemverklebungen, an denen sämtliche Prozesse des Systems beteiligt sind.

2. Die Analyse des Systems auf einen Verklemmungszustand muß berücksichtigen, daß sich während der Deadlockanalyse Anforderungen und Besitz an (globalen) Betriebsmitteln ändern kann.

Innerhalb von Betriebssystemen ist es von Bedeutung, auch Systemverklemmungen zu untersuchen, die nur einen Teil der am System beteiligten Prozesse betreffen. Dies ist deshalb wichtig, da die einzelnen Prozesse normalerweise bis auf die Tatsache, daß sie sich um gemeinsam benutzte vom Betriebssystem verwaltete Ressourcen bewerben, unabhängig sind. Es ist also möglich, daß einige Prozesse vollständig abgeschlossen werden können, obwohl eine Teilmenge von Prozessen sich gemäß Def. 5.1 im Deadlock befinden kann. Dann ist es wichtig, diese Teilmenge festzustellen, um den Verklemmungszustand analysieren und beseitigen zu können.

In einem Parallelprogramm eines Benutzers, in dem die einzelnen Prozesse des Systems auf ein gemeinsam zu erreichendes Ziel hinarbeiten, zieht ein partieller Deadlock jedoch unweigerlich einen totalen Deadlock nach sich, da alle Prozesse des Systems voneinander abhängen. Damit ist es tragbar, sich bei der Analyse auf den (einfacher) erkennbaren Zustand der totalen Systemverklemmung zu beschränken.

In Betriebssystemen, die Multiprogramming zulassen, ist für die Vergabe von Betriebsmitteln normalerweise ein eigener Betriebssystemprozeß zuständig, von dem aus auch die Deadlockanalysen durchgeführt werden. Damit ist es nicht möglich, daß sich an den von diesem Prozeß verwalteten Daten zur Beschreibung des momentanen Zustands der Betriebsmittel während der Deadlockanalyse etwas ändern kann.

In den hier betrachteten Systemen, in denen Prozesse auf gemeinsam benutzte Information (Betriebsmittel) ohne den "Umweg" über ein zentrales Element zugreifen, ist es dagegen notwendig, auch für die Deadlockanalyse von dezentralen Formen der Kommunikation und Synchronisation zwischen Prozessen auszugehen.

Daß die Beachtung dieses Gesichtspunktes Analysen aufwendiger gestaltet, sei kurz an einem Beispielsystem erläutert, in dem Betriebsmittelanforderungen durch die beteiligten Prozesse verwaltet werden und die Deadlockanalyse von einem zentralen Analyseprozeß durchgeführt wird.

Angenommen Prozeß P_i habe eine unerfüllbare Forderung bezgl. der Betriebsmittelklasse B_k und teilt dies dem Analyseprozeß mit. In dem Moment, in dem die Unerfüllbarkeit der Forderung von P_i nach B_k notiert ist, gibt ein zweiter Prozeß P_j so viele Einheiten von B_k frei, daß jetzt die Forderung von P_i erfüllbar ist. Unmittelbar danach wird P_j wegen einer (anderen) unerfüllbaren Forderung blockiert. Dann wird P_i , obwohl er in seiner Berechnung fortfahren kann, vom Analyseprozeß als "blockiert" geführt und kann damit eine nicht gerechtfertigte Deadlockvermutung auslösen. Ein solcher Irrtum kann nicht geschehen, wenn während der Analyse auf einen möglichen Deadlock bezgl. der globalen Betriebsmittel ein konstanter Zustand herrscht.

Zur Sicherstellung eines konstanten Betriebsmittelstatus läßt sich natürlich die bei der Analyse vorausgesetzte Situation erzwingen, d. h. die Betriebsmittelzuteilung wird über den gleichen zentralen Prozeß abgewickelt, der auch die Analysen auf mögliche Deadlocks durchführt. Dieses Verfahren ist dann anwendbar, wenn ein Prozessor extra dafür zur Verfügung steht und wenn Anforderungen in solchen Zeitabständen eintreffen, daß durch die Betriebsmittelzuteilung und Deadlockanalyse keine entscheidend die Gesamtausführungszeit des Programms beeinflussende Verzögerung von Anwenderprozessen erfolgt. Da beide Annahmen für die untersuchten Systeme als nicht realistisch angesehen werden können, muß ein anderer Weg zur Sicherung eines konstanten Betriebsmittelstatus gefunden werden.

Eine weitere Möglichkeit, während der Deadlockanalyse einen konstanten Betriebsmittelstatus zu sichern, besteht darin, während der Deadlockanalyse alle Prozesse, die eine Veränderung am gegenwärtigen Betriebsmittelstatus vornehmen wollen, zu blockieren. Abgesehen davon, daß das Blockieren und Deblockieren

einer Menge von Prozessen einen erheblichen Aufwand darstellt, werden im allgemeinen auch Prozesse blockiert, die nicht wegen momentan unerfüllbarer Forderungen warten, sodaß signifikante Verschlechterungen der Gesamtausführungszeit des Programms die Folge sein können.

Beide Verfahren sind deshalb nicht geeignet, weil sie im deadlockfreien Fall das Gesamtsystem in seiner Laufzeit durch Blockieren lauffähiger Prozesse beeinträchtigen.

Hier beschriebene Algorithmen zur Deadlockerkennung gehen davon aus, daß im Deadlockfall über eine längere Zeit ein konstanter Betriebsmittelstatus vorliegt, da ja keiner der am System beteiligten Prozesse wegen Blockade einen Berechnungsschritt vornehmen und damit den Betriebsmittelstatus ändern kann. Während der Deadlockanalyse wird einfach so getan, als läge ein konstanter Betriebsmittelstatus vor. Wird der untersuchte Betriebsmittelstatus als "deadlockfrei" erkannt, bricht das Verfahren ab. Sonst wird der Betriebsmittelstatus gerettet und die gleiche Analyse noch einmal durchgeführt. Wird dann noch einmal das System als im Deadlock befindlich ermittelt und durch Vergleich von gerettetem zu aktuellem Betriebsmittelstatus festgestellt, daß sich seit der letzten Prüfung am Betriebsmittelstatus keine Veränderungen ergeben haben, ist die Deadlocksituation gegeben.

Die einzelnen Algorithmen unterscheiden sich im wesentlichen in der Beschreibung des Status der Globaldaten (Betriebsmittel), die verschiedene Analysetechniken ermöglichen bzw. erfordern.

5.3.2 Erkennung von Systemverklemmungen in Parallelprogrammen mit E/A-orientierter Kommunikation

In Parallelprogrammen, die Informationen mittels Kommunikationsprozeduren über FIFO-Puffer austauschen, kann eine Deadlockanalyse allein aufgrund der Zustände der im System befindlichen Puffer durchgeführt werden. Die einzigen möglichen Wartegründe für Prozesse sind:

1. Eine zu lesende Information ist nicht verfügbar, d. h. der entsprechende Puffer ist leer; der Prozeß muß so lange warten, bis ein anderer Prozeß den Puffer beschrieben hat (Lesekonflikt).
2. Eine zu schreibende Information kann nicht im Puffer abgelegt werden, da dieser zu voll ist; der Prozeß muß warten, bis ein anderer Prozeß genügend Elemente ausgelesen hat (Schreibkonflikt).

Der Status des Betriebsmittels "Puffer" läßt sich durch folgende Größen beschreiben:

FREI	Zahl der freien Elemente
BELEGT	Zahl der belegten Elemente
LESEFORDERUNG	Zahl der aus dem Puffer zu lesenden Elemente
SCHREIBFORDERUNG	Zahl der in den Puffer zu schreibenden Elemente

Ein Lesekonflikt besteht dann, wenn

LESEFORDERUNG > BELEGT,

ein Schreibkonflikt dann, wenn

SCHREIBFORDERUNG > FREI.

Diese Form der Beschreibung ermöglicht es jedoch nicht, Aussagen bezgl. konstanter Betriebsmittelzustände während der Deadlockanalyse zu machen. So lassen sich die Zustände "Puffer nicht verändert" und "Ein Element gelesen, ein Element geschrieben" anhand dieses Formalismus nicht unterscheiden. (Im

ersten Fall wäre eine Deadlockvermutung richtig, im zweiten nicht in jedem Fall.) Wenn jedoch die Zahl der gelesenen und geschriebenen Elemente als Grundlage zur Beschreibung verwendet wird, können die beiden Zustände unterschieden werden. Dies ist ein zusätzliches Argument für die Gestaltung der Synchronisation über Ereigniszähler statt über Semaphore.

Unter diesen Voraussetzungen kann der in Abb. 5.9 dargestellte Algorithmus zur Deadlockanalyse verwendet werden.

Die dabei verwendeten Größen haben folgende Bedeutung:

DURCHLAUF	Nummer der Analyse (1: erste Prüfung, 2: Prüfung auf Veränderungen am Betriebsmittelstatus)
ANZPROZ	Zahl der im System befindlichen Prozesse
ANZBLOCK	Zahl der im System blockierten Prozesse
ANZPUF	Zahl der im System befindlichen Puffer
READANZ _i	Zahl der aus Puffer i gelesenen Elemente
LASTREAD _i	Zahl der aus Puffer i gelesenen Elemente bei der letzten Deadlockanalyse
WRITEANZ _i	Zahl der in Puffer i geschriebenen Elemente
LASTWRITE _i	Zahl der in Puffer i geschriebenen Elemente bei der letzten Deadlockanalyse
GROESSE _i	Zahl der Elemente, die Puffer i speichern kann
LESEFORDERUNG _i	Zahl der von Puffer i zu lesenden Elemente
SCHREIBFORDERUNG _i	Zahl der in Puffer i zu schreibenden Elemente

Wenn Analysen auf Systemverklemmungen zyklisch durchgeführt werden (z. B. durch einen eigenen Prozeß oder durch den Kontrollprozessor des hierarchischen Multiprozessorsystems in Wartesituationen) kann einfach gezeigt werden:

1. Wenn DEADLOCKDETECTION den Wert DEADLOCK zurückliefert, befindet sich das System im Verklemmungszustand.

```

SUBR DEADLOCKERKENNUNG = () INT
BEGIN

    /*Zustandsanalyse*/

    ANZBLOCK := 0;
    FOR I TO ANZPUF
    DO BELEGT := WRITEANZ[I] - READANZ[I];
       FREI := GROESSE[I] - BELEGT;
       IF LESEFORDERUNG[I] > BELEGT /*Lesekonflikt*/
       OR SCHREIBFORDERUNG[I] > FREI /*Schreibkonflikt*/
       THEN ANZBLOCK := ANZBLOCK + 1
       FI
    OD;
    IF ANZBLOCK < ANZPROZ
    THEN DURCHLAUF := 1; RETURN (NO DEADLOCK)
    FI;

    /*Deadlockvermutung*/

    IF DURCHLAUF = 1
    THEN DURCHLAUF := DURCHLAUF + 1;
       FOR I TO ANZPUF
       DO LASTREAD[I] := READANZ[I];
          LASTWRITE[I] := WRITEANZ[I]
       OD
       RETURN(NO DEADLOCK)
    FI;

    /*Überprüfung der Deadlockvermutung*/

    FOR I TO ANZPUF
    DO IF LASTREAD[I] /= READANZ[I]
       OR LASTWRITE[I] /= WRITEANZ[I]
       THEN DURCHLAUF := 1; RETURN(NO DEADLOCK)
       FI
    OD;
    RETURN(DEADLOCK)

END;

```

Abb. 5.9: Algorithmus zur Erkennung von Systemverklemmungen in Parallelprogrammen mit einer Kommunikation über FIFO-Puffer

2. Wenn sich das System in einem Verklemmungszustand befindet, wird eine Situation erreicht, in der nach Ausführung von DEADLOCKDETECTION der Wert DEADLOCK zurückgeliefert wird.

Theoretisch ist zwar eine (fehlerhafte) Struktur in der Kommunikation zwischen Prozessen konstruierbar, in der obige Analyse beliebig oft durchlaufen wird. Ein solcher Fall ist jedoch wenig wahrscheinlich und in bisher durchgeführten Implementierungen noch nicht aufgetreten.

Ein Vorzug dieses Algorithmus und seiner Umgebung ist, daß nach der Deadlockerkennung unmittelbar eine "systemorientierte" Diagnose über die Ursachen der Verklemmung durchgeführt werden kann (vgl. Kap. 5.3.5). Nachteil ist jedoch, daß er nur innerhalb dieser speziellen Programmierumgebung (Kommunikation zwischen Prozessen über FIFO-Puffer mit einer Synchronisation über Ereigniszähler) anwendbar ist. Wie Erweiterungen zur Erkennung von Systemverklemmungen aussehen können, die aus beliebig formulierten Blockiert-Bedingungen entstehen, wird in den beiden nachfolgenden Kapiteln diskutiert.

5.3.3 Erkennung von Systemverklemmungen in Parallelprogrammen mit normierten Blockiert-Bedingungen

Beim in Kap. 5.3.2 vorgestellten Algorithmus zur Erkennung einer Systemverklemmung in Parallelprogrammen werden zur Analyse Belegungszustände von Puffern mit den Anforderungen an freien oder belegten Pufferelementen verglichen und damit die Erfüllbarkeit von Anforderungen festgestellt. Eine Systemverklemmung liegt dann vor, wenn die Zahl der nicht erfüllbaren Anforderungen gleich der Zahl der im System befindlichen Prozesse ist, wobei die Voraussetzung gilt, daß kein Prozeß mehr als eine Anforderung stellt.

Dieses Puffermodell läßt sich auch auf die Erfüllung beliebiger Bedingungen für einen Programmfortschritt auf einem Prozessor eines Multiprozessorsystems anwenden. Ein Prozeß wartet auf die Erfüllung einer momentan nicht gewährleisteten Bedingung, indem er Veränderungen an den Parametern der Bedingung einem Puffer entnimmt, der von dem diese Parameter ändernden Prozeß entsprechend beschrieben wird. Warten auf die Erfüllung einer Bedingung wird in diesem Fall auf das Warten auf Vorhandensein von (neuen) Bedingungsparametern zurückgeführt.

In Multiprozessorsystemen, bei denen von einem Prozessor direkt auf Daten eines anderen Prozessors zugegriffen werden kann, erscheint dieses Verfahren jedoch umständlich. Wenn sich Blockiert-Bedingungen immer in der Form

"Warte, bis eine (von einem fremden Prozessor veränderbare) Referenzvariable einen bestimmten Sollwert erreicht oder überschritten hat"

formulieren lassen, läßt sich das Verfahren des Algorithmus der Abb. 5.9 entsprechend anpassen. Dazu ist es lediglich notwendig, die Größen "Adresse von Referenzwert", "Sollwert" und "zu erfüllende Bedingung zum Programmfortschritt" an fester Stelle zu hinterlegen, damit sie vom Analyseprozeß ausgewertet werden können.

Um zwei verschiedene Wartezustände eines Prozesses unterscheiden zu können, ist jetzt jedoch ein zusätzlicher Ereigniszähler notwendig, da nicht vorausgesetzt werden kann, daß sich in verschiedenen Wartezuständen die den Wartezustand beschreibenden Größen voneinander unterscheiden. Unterschiedliche Wartezustände können in diesem Fall erkannt werden, wenn der Ereigniszähler bei Eintritt in einen Wartezustand oder bei Verlassen eines Wartezustands (ähnlich wie READANZ oder WRITEANZ) inkrementiert wird.

Unter diesen Voraussetzungen kann der in Abb. 5.10 dargestellte Algorithmus zur Erkennung einer Systemverklemmung verwendet werden.

Die verwendeten (prozeßspezifischen) Größen haben dabei folgende Bedeutung:

REFWERT	Referenz auf die vom fremden Prozeß zu ändernde Variable
SOLLWERT	Wert, den REFWERT gemäß BEDINGUNG zum Programmfortschritt annehmen muß
BEDINGUNG	Bedingung zum Programmfortschritt (z. B. -1: Warten, bis REFWERT < SOLLWERT 0: Warten, bis REFWERT = SOLLWERT 1: Warten, bis REFWERT > SOLLWERT)
ZUSTAND	Prozeßzustand: blockiert oder laufend
EVENTCOUNT	Zahl der bislang erreichten bzw. verlassenen Wartezustände
LASTEVT	Zahl der bislang erreichten bzw. verlassenen Wartezustände bei der letzten Deadlockanalyse

Wie in Kap. 5.3.2 läßt sich bei zyklischem Aufruf von DEADLOCKDETECTION beweisen, daß bei Rückgabe der Quittung DEADLOCK sich das Prozeßsystem im Verklemmungszustand befindet.

Parallelprogramme mit einer alleinigen Kommunikation der Prozesse über FIFO-Puffer lassen sich auch mit diesem Algorithmus auf einen Verklemmungszustand hin überprüfen, wenn ein Lesekonflikt über

"Warten, bis WRITEANZ > READANZ+LESEFORDERUNG-1"

und ein Schreibkonflikt über die Bedingung

"Warten, bis READANZ > WRITEANZ-GROESSE+SCHREIBFORDERUNG-1"

```

SUBR DEADLOCKDETECTION = () INT
BEGIN
    /*Zustandsanalyse*/
    FOR I TO ANZPROZ
    DO IF ZUSTAND[I] = LAUFEND
        THEN DURCHLAUF := 1; RETURN(NO DEADLOCK)
        FI;
        CASE BEDINGUNG[I] EQUAL -1, 0, 1
        IN $(IF REFWERT[I] < SOLLWERT[I]
            THEN DURCHLAUF := 1; RETURN(NO DEADLOCK)
            FI$),
            ...
        /*für andere Bedingungen entsprechend*/
        ...
    ESAC
    OD;

    /*Deadlockvermutung*/

    IF DURCHLAUF = 1
    THEN FOR I TO ANZPROZ
        DO LASTEVT[I] := EVENTCOUNT[I]
        OD;
        DURCHLAUF := DURCHLAUF + 1;
        RETURN(NO DEADLOCK)
    FI;

    /*Überprüfung der Deadlockvermutung*/

    FOR I TO ANZPROZ
    DO IF LASTEVT[I] /= EVENTCOUNT[I]
        THEN DURCHLAUF := 1; RETURN(NO DEADLOCK)
        FI
    OD;
    RETURN(DEADLOCK)
END;

```

Abb. 5.10: Algorithmus zur Erkennung von Systemverklemmungen in Parallelprogrammen mit normierten Blockiert-Bedingungen

jeweils auf den entsprechenden Puffer bezogen formuliert wird. Ein prozeßspezifischer Ereigniszähler kann mit Hilfe der Summe der Größen READANZ und WRITEANZ der Puffer definiert werden, bezgl. derer der jeweilige Prozeß Anforderungen stellt.

5.3.4 Erkennung von Systemverklemmungen in Parallelprogrammen mit beliebigen Blockiert-Bedingungen

Bei den bislang vorgestellten Algorithmen zur Erkennung von Systemverklemmungen in Parallelprogrammen wurde vorausgesetzt, daß Blockiert-Bedingungen in einer normierten Form formuliert werden und vom die Deadlockanalyse durchführenden Prozeß ausgewertet werden können.

Dies ist für beliebige Parallelprogramme eine zu starke Einschränkung. Zum einen kann eine Blockiert-Bedingung aus mehreren Bedingungen bestehen, von denen zum Programmfortschritt alle oder nur bestimmte Teile erfüllt sein müssen. Zum anderen kann ein Prozessor auf Veränderungen an Daten eines fremden Prozessors warten, auf dessen Daten der die Deadlockanalyse durchführende Prozessor nicht zugreifen kann.⁵⁴⁾

Eine Erkennung eines konstanten Verklemmungszustandes ist in diesem Fall möglich, wenn die Blockiert-Bedingung von den blockierten Prozessen mehrfach überprüft wird und die Zahl der Überprüfungen mit dem Ergebnis "blockiert" so vermerkt wird, daß sie vom die Deadlockanalyse durchführenden Prozeß ausgewertet werden kann. Auf dieser Grundlage baut der nachfolgend beschriebene Algorithmus zur Erkennung von Systemverklemmungen in Parallelprogrammen für ein hierarchisches Multiprozessorsystem auf.

54) Dies gilt für alle Multiprozessorsysteme, bei denen es keinen Prozessor gibt, der den gesamten für Informationsaustausch zwischen Prozessoren verfügbaren Speicherraum erreichen kann.

Im folgenden soll zunächst davon ausgegangen werden, daß der die Deadlockanalyse durchführende Prozeß (Analyseprozeß) Zugriff auf alle einen Blockiert-Zustand beschreibenden Größen aller Prozesse im System hat. Anschließend wird kurz erläutert, wie der Algorithmus für beliebige hierarchische Multiprozessorsysteme einfach erweitert werden kann.

Pro Prozessor seien die folgenden Größen definiert:

ZUSTAND:	Zustand des Prozessors mit den Werten
AKTIV	Prozessor arbeitet (Initialisierungswert)
BLOCKIERT	Prozessor arbeitet nicht, weil eine Bedingung zum Programmfortschritt nicht erfüllt ist
VERKLEMMT	Prozessor arbeitet nicht und ist an einer Systemverklemmung beteiligt
FREECOUNT	Zahl der verlassenen Wartezustände (initialisiert mit 0)
LASTFREE	Wert von FREECOUNT bei der letzten Deadlockanalyse
WAITCOUNT	Zahl der Überprüfungen der Blockiert-Bedingungen mit Ergebnis "blockiert" (initialisiert mit 0)
LASTWAIT	Wert von WAITCOUNT bei der letzten Deadlockanalyse

Im Blockiert-Zustand möge ein Prozessor folgendes Programmstück durchlaufen:

```
WHILE "Bedingung zum Programmfortschritt nicht erfüllt"
DO QUITTUNG := DELAY(); /*Verzögertes Aktives Warten*/
... /*Auswerten der Quittung*/
OD;
CONTINUE(); /*Beendigung des Wartezustandes melden*/
```

Für nicht die Deadlockanalyse durchführende Prozessoren sind die Routinen DELAY und CONTINUE wie folgt definiert:

```
SUBR DELAY = () INT
BEGIN
  IF ZUSTAND = AKTIV
  THEN ZUSTAND := BLOCKIERT
  ELSE IF ZUSTAND = VERKLEMMT THEN RETURN (DEADLOCK) FI
      WAITCOUNT := WAITCOUNT + 1
  FI;
  VERZOEGERUNG (); /*vgl. Kap. 5.2.2*/
  RETURN (NO DEADLOCK)
END;

SUBR CONTINUE = ()
BEGIN
  ZUSTAND := AKTIV;
  FREECOUNT := FREECOUNT + 1;
  WAITCOUNT := 0
END;
```

Der die Deadlockanalyse durchführende Prozessor hält in der Variablen DURCHLAUF fest, wie oft hintereinander er sämtliche erreichbaren Prozesse des Systems als "blockiert" erkannt hat (Initialwert 1). Innerhalb der Routine DELAY kann nun eine Deadlockanalyse wie in Abb. 5.11 dargestellt durchgeführt werden. Die Routine CONTINUE für den Analyseprozessor besteht lediglich aus der Anweisung

```
DURCHLAUF := 1,
```

um für folgende Deadlockanalysen wieder einen "Initialisierungszustand" zu gewährleisten.

Analog zu den Algorithmen der Kapitel 5.3.2 und 5.3.3 wird bewiesen:

1. Wenn sich das System im Deadlock befindet, wird einmal die Routine DELAY über den Ausgang "Deadlock" verlassen.
2. Wenn die Routine DELAY über den Ausgang "Deadlock" verlassen wird und die Analyse alle Prozesse des Systems umfaßt, befindet sich das System im Deadlock.

```

SUBR DELAY = () INT
BEGIN

    /*Sind alle erreichbaren Prozessoren
       des Systems im Wartezustand?*/

    FOR I TO ANZPROZ
    DO IF ZUSTAND[I] = AKTIV
        THEN DURCHLAUF := 1; RETURN (NO DEADLOCK)
        FI
    OD;

    /*Alle erreichbaren Prozesse im Wartezustand. Wurde
       DELAY mit diesem Ergebnis schon einmal durchlaufen?*/

    IF DURCHLAUF = 1
    THEN DURCHLAUF := DURCHLAUF + 1;
        FOR I TO ANZPROZ
        DO LASTFREE[I] := FREECOUNT[I];
            LASTWAIT[I] := WAITCOUNT[I]
        OD;
        RETURN (DEADLOCKVERMUTUNG)
    FI;

    /*Wurde beim zweiten Mal der gleiche
       Wartezustand analysiert?*/

    FOR I TO ANZPROZ
    DO IF LASTFREE[I] < FREECOUNT[I]
        THEN DURCHLAUF := 1; RETURN (NO DEADLOCK)
        /*neuer Wartezustand*/
        FI;
        IF LASTWAIT[I] >= WAITCOUNT[I]
        THEN RETURN (DEADLOCKVERMUTUNG)
        /*Prozessor hat seine Blockiert-Bedingung
           seit dem letzten Durchlauf nicht neu
           geprüft; keine Aussage möglich*/
        FI
    OD;
    FOR I TO ANZPROZ DO ZUSTAND[I] := VERKLEMMT OD;
    RETURN(DEADLOCK)

END;

```

Abb. 5.11: Algorithmus zur Erkennung von Systemverklemmungen in Parallelprogrammen mit beliebigen Blockiert-Bedingungen

Beweis:

1. Das System befindet sich im Deadlock, d. h. Analyseprozessor und alle übrigen Prozessoren sind im Wartezustand.

Der Analyseprozessor erkennt beim ersten Durchlauf durch DELAY (DURCHLAUF=1), daß alle erreichbaren Prozessoren im Wartezustand sind. Damit hält der Analyseprozessor für alle anderen erreichbaren Prozessoren den letzten Ereignisstand (FREECOUNT, WAITCOUNT) fest und verläßt die Routine über den Ausgang "Deadlockvermutung".

Beim zweiten Durchlauf durch DELAY (DURCHLAUF>1) erkennt der Analyseprozessor, daß sich an den Größen FREECOUNT der anderen Prozesse nichts geändert hat, da die übrigen Prozessoren ihre Blockierschleife nicht verlassen können. (Entsprechendes gilt übrigens auch für alle weiteren Durchläufe.)

Wenn jeder Prozessor nach endlicher Zeit eine Anweisung ausführt und somit innerhalb der Warteschleife WAITCOUNT ändert, stellt der Analyseprozessor bei einem der nächsten Durchläufe durch DELAY fest, daß sich zwar etwas an WAITCOUNT, nicht jedoch an FREECOUNT für die anderen Prozessoren geändert hat.

Damit wird die Routine DELAY über den Ausgang "Deadlock" verlassen.

2. Die Routine DELAY des Analyseprozessors wird über den Ausgang "Deadlock" verlassen.

Daraus folgt:

1. Die Routine wurde mindestens zweimal durchlaufen.
2. In beiden Fällen wurde festgestellt, daß sich alle Prozessoren des Systems im Wartezustand befinden.
3. Beim zweiten oder einem folgenden Durchlauf wurde festgestellt, daß sich für alle Prozessoren bezgl. WAITCOUNT eine Änderung und bezgl. FREECOUNT keine Änderung vollzogen hat.

Damit ist gezeigt, daß zwischen beiden Durchläufen von DELAY jeder vom Analyseprozessor erreichbare Prozessor seine Blockiert-Bedingung überprüft und die Warteschleife nicht verlassen hat, sich also immer noch im Wartezustand befindet. (Sonst hätte der Analyseprozessor DELAY über irgendeinen anderen Ausgang verlassen.)

Es ist auch nicht möglich, daß beim zweiten oder einem der folgenden Durchläufe durch DELAY ein neuer Wartezustand analysiert wird, da in diesem Fall eine Änderung mindestens eines FREECOUNT stattgefunden hätte und somit der Analyseprozessor die Routine an einer anderen Stelle verlassen hätte.

Es ist auch nicht möglich, daß die Blockiert-Bedingung für irgend einen der erreichbaren Prozessoren zwischen zwei Durchläufen aufgehoben wurde; denn dadurch, daß der Analyseprozessor über den Vergleich von WAITCOUNT und LASTWAIT festgestellt hat, daß jeder andere erreichbare Prozessor die Blockiert-Bedingung überprüft und die Warteschleife nicht verlassen hat, ist sichergestellt, daß die Blockiert-Bedingung noch immer gilt. Da die Deadlockanalyse sämtliche Prozesse des Systems umfaßt, hätte die Aufhebung einer Blockiert-Bedingung nur von einem der analysierten Prozesse veranlaßt werden können. Dieser hätte aber seinerseits die Warteschleife verlassen müssen und dabei seine Größe FREECOUNT in für den Kontrollprozessor feststellbarer Weise erhöhen müssen. In diesem Fall hätte der Analyseprozessor die DELAY-Routine jedoch nicht über den Ausgang "Deadlock" verlassen.

Damit ist gezeigt, daß jeder erreichbare Prozessor des Systems im Wartezustand ist und diesen mindestens zweimal überprüft hat, ohne ihn zu verlassen, und sich auch zwischen zwei Überprüfungen an der Gültigkeit der Blockiert-Bedingungen nichts geändert hat, d. h. ein konstanter Wartezustand bezgl. aller Prozessoren erreicht ist.

Wenn mit der Deadlockanalyse alle Prozessoren des Systems erfaßt werden, befindet sich also das System nach Verlassen der Routine DELAY des Analyseprozessors im Deadlock.■

Dieser Algorithmus wurde innerhalb der in Kap. 5.1.1 beschriebenen elementaren parallelen Programmierungsumgebung für das EGPA-System implementiert. Die Deadlockanalyse wird vom Kontrollprozessor (der in diesem Fall sämtliche Prozessoren überprüfen kann) vorgenommen, wenn dieser selbst in einem Wartezustand ist.

Er läßt sich wie folgt für mehrschichtige hierarchische Multiprozessorsysteme wie etwa [FKLV83] erweitern.

Jeder Prozessor hält im Wartezustand fest, wieviele ihm untergeordnete Prozessoren einschließlich ihm selbst im Wartezustand sind. (Für Prozessoren der unteren Ebene ergibt sich der Wert 1, bei darüberliegenden Ebenen wird in Form der Baumhöhenreduktion über die Zahl der untergeordneten wartenden Prozessoren aufsummiert.) Wenn vom Prozessor in der Spitze der Hierarchie eine Deadlockanalyse eingeleitet wird (dies geschieht selbstverständlich nur, wenn ihm sämtliche Prozesse des Systems als im Wartezustand befindlich gemeldet sind), werden über die Schichten hinweg alle Prozessoren von der Deadlockanalyse unterrichtet und führen nun ihrerseits die Prüfung auf Konstanz der Wartezustände bezgl. ihrer untergeordneten Prozessoren durch.

Wenn keiner der Prozesse blockiert wird und wie gezeigt über DELAY und CONTINUE die Wartezustände aktiv durchläuft, erhält der Prozessor in der Spitze der Hierarchie bei einem der späteren Durchläufe durch die Deadlockerkennungsroutine einen Zustand, in dem

1. weniger Prozesse im Wartezustand angezeigt sind als im System vorhanden sind, oder
2. mindestens ein Prozeß in einem neuen Wartezustand ist (Änderung mindestens eines FREECOUNT seit Einleitung der Deadlockanalyse) oder

3. alle Prozesse des Systems ihre Blockiert-Bedingung überprüft und ihren Wartezustand nicht verlassen haben.

Während in den ersten beiden Fällen kein Deadlockzustand vorliegt, ist im dritten Fall ein konstanter Wartezustand für alle Prozesse im System erreicht und damit die Systemverklemmung gegeben.

Nach Erkennung eines Verklemmungszustandes erhebt sich die Frage, wie dieser an den Benutzer weitergegeben werden kann und welche Reaktionen danach von System- und/oder Benutzerseite zum kontrollierten Programmabbruch vorgenommen werden können. Eine Zusammenstellung möglicher Reaktionen wird in Kap. 5.3.5 gegeben.

Die Funktionsfähigkeit dieses Algorithmus hängt natürlich davon ab, daß der Benutzer "diszipliniert" programmiert, z. B. Beginn und Ende von Wartezuständen anzeigt, und daß kein Prozeß beliebig lange stillgelegt wird, d. h. zumindest verzögert aktiv gewartet wird.⁵⁵⁾ In Betriebssystemen (vgl. etwa [Lin81]) wird zur Vermeidung von Systemverklemmungen, die durch "undiszipliniertes" Programmieren entstehen und damit nicht mehr kontrolliert werden können, ein Timeout-Mechanismus implementiert, der nach Ablauf einer bestimmten Zeitspanne an einer Systemverklemmung beteiligte Prozesse zwangsbeendet.

55) Die beiden anderen Algorithmen zur Erkennung von Systemverklemmungen verlangen "nur", daß der Analyseprozeß nicht beliebig lange blockiert wird. Von ähnlichen Voraussetzungen müssen alle Algorithmen zur Deadlockerkennung ausgehen, da sonst das Deadlockproblem nicht entscheidbar ist.

5.3.5 Diagnose und Beseitigung von Systemverklemmungen, Stoppen von Prozessoren

Wenn der Lauf eines Parallelprogramms in einer Systemverklemmung resultierte, reicht es natürlich nicht aus, die Tatsache lediglich zu erkennen. Gefragt sind Maßnahmen zur Diagnose der Ursachen und anschließend zur kontrollierten Beendigung des Programms.

Dies führt auf die Problematik der Behandlung von Fehler- bzw. Ausnahmesituationen in Programmabläufen, wie sie etwa in [Goo75] oder [MeRa77] allgemein diskutiert werden. In [Web83] findet sich eine ausführliche Zusammenstellung von Methoden zur Behandlung von Ausnahmesituationen in Systemen aus parallel ablaufenden Prozessen, bei denen ein Programmabbruch nicht einfach toleriert werden kann.

Hier soll lediglich eine Möglichkeit aufgezeigt werden, wie auf Fehler in Parallelprogrammen reagiert werden kann, die in einer Systemverklemmung resultieren. In Kap. 5.4 werden weitergehende Möglichkeiten aufgezeigt, Fehler in Parallelprogrammen einzugrenzen.

Die Algorithmen zur Erkennung von Systemverklemmungen werden immer dann aufgerufen, wenn eine momentan nicht erfüllte Bedingung einen weiteren Programmfortschritt nicht gestattet und deswegen gewartet werden muß. Sie wurden so gestaltet, daß im Verklemmungsfall eine Fehlerquittung als Ergebnis der Warteroutine zurückgeliefert wird. Es ist also möglich, daß über die Rückgabequittung der Warteroutine an den Benutzer eine Information über einen Verklemmungszustand weitergegeben werden kann. Von dort aus kann dann der Benutzer selbst Maßnahmen zur Diagnose und Beseitigung der Verklemmung einleiten. Abb. 5.12 zeigt, wie eine benutzerseitige Diagnose einer Systemverklemmung im EGPA-System am Beispiel der Textformatierung gestaltet werden kann. (Im vorliegenden Fall wäre die angemessene Reaktion die, zumindest einen der Puffer zu vergrößern, bezgl. denen ein Schreibkonflikt besteht, d. h. S2VL, VLZBF und/oder ZBFSE.)

SYSTEMVERKLEMMUNG:

PROZESSOR B WEGEN SCHREIBKONFLIKT AN PUFFER S2VL BLOCKIERT
(3 VON 3 ELEMENTEN BELEGT, 1 ZUM SCHREIBEN ANGEFORDERT)
PROZESSOR A1 WEGEN SCHREIBKONFLIKT AN PUFFER VLZBF BLOCKIERT
(3 VON 3 ELEMENTEN BELEGT, 1 ZUM SCHREIBEN ANGEFORDERT)
PROZESSOR A2 WEGEN LESEKONFLIKT AN PUFFER VLZT BLOCKIERT
(0 VON 3 ELEMENTEN BELEGT, 1 ZUM LESEN ANGEFORDERT)
PROZESSOR A3 WEGEN LESEKONFLIKT AN PUFFER ZTSE BLOCKIERT
(0 VON 3 ELEMENTEN BELEGT, 1 ZUM LESEN ANGEFORDERT)
PROZESSOR A4 WEGEN SCHREIBKONFLIKT AN PUFFER ZBFSE BLOCKIERT
(3 VON 3 ELEMENTEN BELEGT, 1 ZUM SCHREIBEN ANGEFORDERT)
PROZESSOR B WEGEN LESEKONFLIKT AN PUFFER SEASUS BLOCKIERT
(0 VON 3 ELEMENTEN BELEGT, 1 ZUM LESEN ANGEFORDERT)

Abb. 5.12: Beispiel zur Diagnose einer Systemverklemmung im Anwendungsbeispiel Textformatierung mit einer Zuordnung von Programmfunktionen zu Prozessoren (vgl. Abb. 4.2)

Innerhalb eines Rahmens zur Erstellung von Parallelprogrammen ist es sicher nicht ausreichend, allein über Statusvariable Systemverklemmungen an den Benutzer zu melden, da davon ausgegangen werden muß, daß solche Statusmeldungen ignoriert werden.

Für die in Kap. 5.1.1 und 5.1.3 beschriebenen parallelen Programmierungsumgebungen auf dem EGPA-System wurde eine zweite Ebene zur Behandlung von Fehlersituationen in Form einer Alarmmarke auf dem B-Prozessor (Kontrollprozessor) eingerichtet. Diese (dem Benutzer zugängliche) Alarmmarke wird immer dann angesprungen, wenn auf irgendeinem Prozessor ein Alarm auftritt (z. B. Division durch 0) oder nach Feststellung einer Systemverklemmung ein weiteres Mal eine Warteroutine angesprungen wird. Beim Ansprung dieser Alarmmarke ist immer noch eine Auswertung fehlerrelevanter Daten möglich. Danach werden alle Prozessoren veranlaßt, ihren Lauf abzuschließen. Falls Arbeitsprozessoren noch auf Arbeit warten, wird diesen die Kennung zur Programmbeendigung übermittelt, andernfalls diese mit Hilfe eines Betriebssy-

stemdienstes zwangsbeendet. In jedem Fall wird vom Kontrollprozessor auf die Beendigung der übrigen Prozessoren gewartet. Dieses Verfahren kann auf eine mehrschichtige Hierarchie in der Weise übertragen werden, daß Beendigungsmeldungen oder Beendigungsmaßnahmen von einer Schicht an die nächsttiefere Schicht weitergegeben werden und Prozessoren einer höheren Schicht auf die Beendigung von Prozessoren einer tieferen Schicht warten.

Mit der Implementierung von Parallelprogrammen in der "E/A-orientierten" Form können alle derartigen Maßnahmen innerhalb der Kommunikations- und Synchronisationsmodule auf einer "System"-Ebene ausgeführt werden. Im Fehlerfall wird lediglich eine Fehlerquittung als Ergebnis der gerade ausgeführten Kommunikationsprozedur zurückgeliefert. Die Auswertung solcher Quittungen von Kommunikationsprozeduren und eine angemessene Reaktion im Fehlerfall ist jedoch auch schon für sequentielle Programme notwendig, sodaß sich in diesem Fall auf Benutzerseite kein zusätzlicher Programmieraufwand ergibt.

Wenn Fehler bzw. daraus resultierende Systemverklemmungen erkannt worden sind, ist meist keine so eindeutige Diagnose möglich, wie dies in Abb. 5.12 angezeigt wurde. In der Regel entsteht eine Systemverklemmung auch nicht unmittelbar nach Auftreten eines Fehlers, sondern erst viel später, nachdem im Programm noch einige mehr oder weniger sinnvolle Operationen ausgeführt wurden. Dann ist es notwendig, eine Auswertung der Programmhistorie vorzunehmen, um den Entstehungsort eines Fehlers zu lokalisieren. In Kap. 5.4 werden Möglichkeiten zusammengestellt, auf der Basis E/A-orientierter Parallelprogramme parallel ablaufende Aktivitäten während und außerhalb des eigentlichen Programmlaufes nachzuvollziehen.

5.4. Testmöglichkeiten für Parallelprogramme

Die Problematik der Fehlersuche in auf Multiprozessorsystemen ablaufenden Programmen ist ein Bereich, der bei der Beschreibung von Parallelimplementierungen weitgehend ausgespart wird. Debugging-Systeme, die ein systematisches Testen von Parallelprogrammen ermöglichen, fehlen noch gänzlich.

Die Zusammenstellung des Kap. 5.4.1 zeigt Gründe auf und gibt in Verbindung mit den in den Kap. 5.4.2 diskutierten Ansätzen eines Debugging Hinweise, die bei der Gestaltung von Debugging-Systemen für Multiprozessorsysteme berücksichtigt werden müßten.

Grundlage der Überlegungen sind Parallelprogramme mit E/A-orientierter Kommunikation, die einen "systemnahen" Einbau von Testsystemen erlauben.

5.4.1 Die Problematik von Testsystemen und Testausgaben in Parallelprogrammen

Wenn Programme für den Einzelprozessor in einem Programmlauf ein nicht erwartetes Ergebnis produzieren, werden die Ursachen normalerweise durch Programmtests gefunden. Testsysteme erlauben es, den Programmablauf für den Benutzer transparent zu machen, indem an charakteristischen Stellen in den Programmablauf eingegriffen werden kann. Hilfsmittel sind Testausgaben, Programmstops und von da ausgehend Möglichkeiten, Variableninhalte zu verändern usw.

Diese Verfahren lassen sich auf Parallelprogramme nicht direkt übertragen. Die folgende Zusammenstellung zeigt die wesentlichsten Gründe dafür auf.

1. Gegebenheiten der Hardware:

Fehler innerhalb von Parallelprogrammen können auf Prozessoren entstehen, die keinen Anschluß zur Peripherie haben und somit durch Testausgaben überhaupt nicht ermittelt werden können. Es

ist sehr viel wahrscheinlicher als in Monoprogrammen, daß Programmfehler durch Ausfall von Komponenten der Hardware verursacht werden.

2. Eindeutigkeit des Fehlerortes:

In der Regel laufen auf verschiedenen Prozessoren eines Multiprozessorsystems die gleichen Programme ab. Damit reicht es nicht aus, allein den Inhalt von Variablen zu protokollieren; es ist auch notwendig, den zugehörigen Prozessor mit zu erfassen. Vorhandene Testsysteme lassen sich im allgemeinen nur an einen laufenden Prozeß, nicht jedoch (was erforderlich wäre) an beliebige Teilmengen parallel ablaufender Prozesse anklemmen.

3. Schwierigkeit der Realisierung von Programmstops:

Es ist praktisch nicht auf einfache Weise realisierbar, Teilmengen von Prozessoren von außerhalb an bestimmten oder beliebigen Programmstellen anzuhalten. Wenn nur einzelne Prozessoren gestoppt werden, können andere noch laufende Prozessoren charakteristische Daten manipulieren, sodaß ein konsistenter Zustand zur Analyse nicht erreichbar erscheint.

4. Reproduzierbarkeit von Ergebnissen:

Das systematische Testen von Programmen für den Einzelprozessor geht maßgeblich davon aus, daß Programme, die mehrfach mit den gleichen Eingaben gestartet werden, immer die gleichen Ergebnisse liefern. Dies kann jedoch nicht mehr garantiert werden, wenn der Programmfortschritt auf einem Prozessor vom Programmfortschritt auf einem anderen Prozessor abhängt.

5. Verfälschung der Abfolge von Ereignissen durch Testausgaben:

Wenn nur ein Prozessor oder nur eine eingeschränkte Teilmenge von Prozessoren Testausgaben durchführen kann, muß eine Strategie bestehen, die die Reihenfolge festlegt, in der "gleichzeitig" vorliegende Ausgaben von verschiedenen Prozessoren ausgeschrieben werden. Diese Strategie muß nicht notwendigerweise die zeitliche Reihenfolge widerspiegeln, in

der zu protokollierende Ereignisse auf verschiedenen Prozessoren entstehen. Ein Effekt, der weitaus unangenehmer ist, ist der, daß durch Testausgaben eine zeitliche Beeinflussung des Programmfortschritts auf einzelnen Prozessoren stattfindet. Diese kann so weit gehen, daß Programme mit eingebauten Testausgaben fehlerfrei ablaufen, während die gleichen Programme ohne Testausgaben den (dann nicht mehr diagnostizierbaren) Fehler produzieren.

Nachfolgend werden für Parallelprogramme mit E/A-orientierter Kommunikation Möglichkeiten vorgeschlagen, den angeschnittenen Problemen zu begegnen:

5.4.2 Zur Realisierung von Testsystemen für Parallelprogramme

Eines der Hauptprobleme bei der Analyse von Parallelprogrammen rührt daher, daß keine Möglichkeit besteht, gleiche Programmabfolgen im Zusammenspiel verschiedener Prozessoren beliebig häufig zu reproduzieren. Dies ist dann nicht von Belang, wenn die Entstehung eines zu untersuchenden Fehlerzustandes genügend genau protokolliert wurde, sodaß aus der Analyse des Ereignisprotokolls Rückschlüsse gezogen werden können.

Multiprozessorsysteme wie C.mmp [HaWu78], CM* [GJS83] oder EGPA [FHHJ83] besitzen im allgemeinen hardwareseitig zusätzliche Einrichtungen (Meßfühler, Diagnoseprozessoren), die während des Laufs eines Parallelprogramms charakteristische Ereignisse wie Pagefaults, Senden von Botschaften, Zugriffe auf Fremdspeicher usw. aufzeichnen können. Ein anderer Weg, ein (mehr benutzerorientiertes) Ereignisprotokoll eines parallelen Programmlaufs zu gewinnen, geht über die Verwendung von Software-Meßsystemen [Klei82]. Aufzuzeichnende Ereignisse werden über Unterprogrammaufrufe an das Meßsystem übermittelt, welches die Ereignisse zusammen mit dem Zeitpunkt ihres Auftretens puffert und bei gefülltem Meßpuffer ausschreibt.

Die Verwendung solcher Systeme hat im Gegensatz zur Verwendung von Testausgaben den Vorteil, daß durch die Protokollierung eines Ereignisses praktisch kaum Zeit verloren geht. Dadurch ergeben sich gegenüber Programmläufen ohne Ereignisprotokollierung auch keine signifikanten Verschiebungen in den Zeitabläufen von Programmen auf verschiedenen Prozessoren. Ein gewisser Nachteil besteht darin, daß es zusätzlicher Programme bedarf, Ereignisprotokolle in für den Benutzer lesbarer Form zu interpretieren. In [Müh84] wird eine Möglichkeit beschrieben, Protokolle von Hardwaremessungen und Softwaremessungen in eine "benutzerfreundliche" Form umzusetzen.

Erfahrungen haben gezeigt, daß die anschaulichste Art der Darstellung von Ereignisprotokollen die über graphische Ausgaben ist (vgl. hierzu [Klei82], [Knei82], [Müh84], [Rat83]).

Für Parallelprogramme mit E/A-orientierter Kommunikation bieten sich als Ereignisse zur Protokollierung die Kommunikationsprozeduren in Verbindung mit den Programmmuständen an, in denen Prozessoren auf andere Prozessoren warten. Da dies die einzigen Stellen sind, an denen Kommunikation zwischen Prozessoren stattfindet, reicht die Analyse einer mit diesen Ereignissen gebildeten Spur aus, um Synchronisationsfehler und das Entstehen von Systemverklemmungen während des Programmlaufs verfolgen zu können. Das Prinzip zur Konstruktion dieses Typs von Programmen (vgl. Kap. 2 und Kap. 5.2) erlaubt es, Maßnahmen, die vor Aufnahme einer solchen Ereignismessung durchzuführen sind, vom eigentlichen Algorithmus weg in eine "Systemebene" (die der Kommunikations- und Synchronisationsmodule) zu verlagern.

Da sich gemäß der Modulhierarchien der Abb. 2.18 und 5.5 in sequentiellen und parallelen Implementierungen Transformations- und Hilfsmodule (in denen der eigentliche Kern des Algorithmus realisiert ist) nicht unterscheiden, kann bis auf Parallelisierungsstrategie und Prozessorkommunikation ein Programmtest in Sequentiellen erfolgen, wo für den Anwender eine wesentlich bessere Testumgebung verfügbar ist.

Doch auch für eigentliche "Parallelteile" läßt sich in Monorechnern ein Programmtest realisieren [Rat83], wenn die vorhandene Programmierungsumgebung einen Ablauf separat programmierbarer Prozesse zuläßt. Eine derartige Ausführungsform eines Parallelprogramms gestattet es auch, Testausgaben von beliebigen "Prozessoren" aus vorzunehmen.

Zur Implementierung eines Parallelprogramms auf einem Monorechner wird jeder Prozessor des für die Implementierung zugrundegelegten Systems durch einen eigenen Prozeß nachgebildet. Für den Programmlauf auf dem Monorechner wird eine Serialisierung der eigentlich parallel ablaufenden Prozesse vorgenommen. Zur Serialisierung parallel ablaufender Prozesse läßt sich eine Reihe von Varianten angeben, die je nach dem resultierenden Prozeßablauf in deterministische und zufällige unterschieden werden können.

Eine deterministische Prozeßsteuerung [Rat83] besteht darin, die einzelnen Prozesse des Prozeßsystems in einer festen Reihenfolge so lange laufen zu lassen, bis sie wegen einer momentan nicht erfüllbaren Bedingung blockiert werden und dann einen Prozeßwechsel gemäß einer festgelegten Abfolge vorzunehmen. Auf dieser Basis wurde die in Kap. 5.1.2 beschriebene elementare parallele Programmierungsumgebung auf der Basis des UNIX-Systems implementiert.

Eine andere Möglichkeit einer deterministischen Prozeßsteuerung läßt sich in Anlehnung an [KaMQ77] realisieren.⁵⁶⁾ Zunächst wird der Prozeß gestartet, der das endgültige Ergebnis anliefert. Dieser läuft so lange, bis er wegen einer noch nicht verfügbaren Eingabe blockiert wird. Dann wird der Prozeß gestartet, der diese Eingabe liefert usw. Ein Prozeßwechsel geschieht immer dann, wenn

56) In der in [KaMQ77] beschriebenen Variante zur Serialisierung paralleler Prozesse wird von unendlich großen Puffern zur Aufnahme von Zwischenergebnissen ausgegangen, sodaß dort kein Schreibkonflikt auftreten kann. Mit Programmabläufen unter dieser Voraussetzung läßt sich auf experimentellem Wege bestimmen, welche Mindestgröße Puffer zur Aufnahme von zwischen Prozessen auszutauschender Information haben müssen.

ein Prozeß wegen einer nicht verfügbaren Eingabe oder einer nicht absetzbaren Ausgabe blockiert wird oder wenn eine Eingabe verbraucht bzw. Ausgabe erzeugt wird, aufgrund derer ein bislang blockierter Prozeß weiter arbeiten kann. Im ersten Fall wird ein Prozeßwechsel auf den Prozeß vorgenommen, der den Blockierungszustand auflösen kann. Im zweiten Fall wird beim Prozeßwechsel der bislang blockierte Prozeß gestartet.

Deterministische Varianten zur Serialisierung paralleler Prozesse reichen aus, beliebig reproduzierbare (systematische) Fehler in Parallelprogrammen aufzuspüren und zu analysieren. Sie gestatten auch, in sehr einfacher Weise eine Analyse im Hinblick auf Systemverklemmungen vorzunehmen. So liegt in der Variante von [Rat83] eine Systemverklemmung dann vor, wenn nach einem Durchlauf über alle Prozesse des Systems festgestellt wird, daß kein Prozeß einen Wartezustand verlassen konnte. In der Variante in Anlehnung an [KaMQ77] liegt eine Systemverklemmung vor, wenn nach Erreichen eines Blockiert-Zustandes ein Prozeß aktiviert wird, der sich selbst im Wartezustand befindet.

Es ist jedoch mit einer Simulation paralleler Prozesse über deterministische Prozeßsteuerungen nicht möglich, sporadisch auftretende nicht beliebig reproduzierbare (vom Laufzeitverhalten der beteiligten Prozesse abhängige) Fehler zu erkennen. Um solche Fehler lokalisieren zu können, kann eine zufällig erfolgende Serialisierung der parallelen Prozesse in Verbindung mit der Auswertung eines dazu gehörigen Ereignisprotokolls ausreichen.

Eine Möglichkeit [Rat83] besteht darin, die Prozesse des Systems nicht in Abhängigkeit des Erreichens oder Verlassens von Blockiert-Zuständen laufen zu lassen, sondern über feste Zeitscheiben hinweg oder über eine bestimmbare Zahl von Anweisungen. In diesem Fall ergibt sich zwar die in Kap. 5.4.1 angeschnittene Problematik bezgl. der Reproduzierbarkeit der Ergebnisse von (fehlerhaften) parallelen Programmläufen. Eine Analyse erscheint auf dieser Ebene jedoch leichter als im realen Multiprozessorsystem durchführbar, da die einzelnen Prozesse einfacher kontrolliert werden können und von jedem Prozeß aus Testausgaben verfügbar gemacht werden können.

Ein sicher besserer Ansatz zur Erzielung einer zufälligen serialisierten Abfolge paralleler Prozesse besteht darin, einen zufällig ausgewählten Prozeß so lange laufen zu lassen, bis er an eine Stelle kommt, an der er blockiert wird oder Daten verändert, die andere Prozesse in ihrem Programmfortschritt beeinflussen können. Dort sollte eine Unterbrechung des laufenden Prozesses und anschließend ein zufälliger (oder vom Benutzer bestimmbarer) Wechsel auf den gleichen oder einen anderen Prozeß erfolgen. Wenn eine solche zufällig entstandene Prozeßabfolge aufgezeichnet wird, kann diese zu späteren Zeitpunkten auch beliebig reproduziert und dabei genauer analysiert werden. In Parallelprogrammen mit E/A-orientierter Kommunikation können alle derartigen Unterbrechungsstellen innerhalb von Kommunikations- und Synchronisationsmodulen eingebaut werden.

Da es schwierig erscheint, für eine Fehleranalyse von Parallelprogrammen wünschenswerte Eigenschaften (vgl. hierzu auch [Wen78], pp. 114-116) in realen Multiprozessorsystemen zu implementieren, stellt dieser Ansatz eines Debugging von Parallelprogrammen auf der Ebene des Monorechners eine durchaus beachtenswerte Alternative dar. Ein weiteres Eingehen auf Probleme des Debugging von Parallelprogrammen vor allem im Hinblick auf die Gestaltung von Codegeneratoren, Programmbindern und -ladern würde den Rahmen dieser Arbeit sprengen.

6 Zum Entwurf einer E/A-orientierten parallelen Programmierungsumgebung

In Kap. 3 wurden parallele Verarbeitungsstrategien für ordnungserhaltende Programmsysteme vorgestellt, in Kap. 5 wurden Aspekte der Implementierung der resultierenden Parallelprogramme diskutiert. Dabei wurden die beiden Linien der Entwicklung zur Realisierung der Kommunikation zwischen den verschiedenen gleichzeitig ablaufenden Prozessen im Parallelprogramm aufgezeigt: Kommunikation über Variable in einem gemeinsam benutzten Speicher und Kommunikation über Botschaften.

In diesem Abschnitt sollen ausgehend von der "E/A-orientierten" Methode zur Implementierung von Parallelprogrammen und den parallelen Verarbeitungsstrategien des Kap. 3 eine parallele Programmstruktur sowie geeignete Sprachmittel definiert und Möglichkeiten der Einbettung in verschiedene Organisationsformen für Multiprozessorsysteme untersucht werden.

6.1 Struktur von Parallelprogrammen, Gestaltungsgrundlagen

Eine Programmierungsumgebung für die Programmentwicklung auf speziellen Multiprozessorsystemen kann anders als eine Programmiersprache mit Hilfsmitteln zur Formulierung parallel verlaufender Aktivitäten den Anwender nicht über die Systemkonfiguration (Prozessoren, Speicher und ihre Verbindung untereinander) im unklaren lassen. Es muß die Möglichkeit bestehen, über bestimmte Kennungen wie Prozessornummern o. ä. bestimmte Teile des Systems gezielt ansprechen zu können. Dies darf jedoch aus Aufwandsgründen nicht dazu führen, daß für jeden Prozessor die Erstellung eines individuellen Programms erforderlich wird.

Wenn das zugrundeliegende Multiprozessorsystem bezgl. Prozessoren, Speichern und ihrer Verbindungen eine reguläre Struktur hat (wie z. B. das hier als Grundlage verwendete EGPA-System), läßt sich eine prozessorabhängige Programmsteuerung über wenige prozessorspezifische Programmbausteine verwirklichen (vgl. [Knei82], [Rat83]). In diesem Fall läßt sich das dem Hardwareentwurf des EGPA-Systems zugrundeliegende Prinzip der konstanten lokalen Komplexität von Prozessorelementen bei "beliebiger" Erweiterbarkeit [Hän82] voll in deren Programmierung übernehmen.

Der in der Modulhierarchie der Abb. 5.5 resultierende Ansatz bei der Entwicklung von Parallelprogrammen legt es nahe, prozessorspezifische Informationen aus dem Kern des Algorithmus (d. h. konkret der Ebene der Transformations- und Hilfsmodule) heraus in Module mit organisationsorientierten Aufgaben (Strategiemodule, Kommunikationsmodule, Synchronisationsmodule) zu verlagern.

Die Analyse zahlreicher Programme im EGPA-System hat gezeigt, daß Programme auf den einzelnen Prozessoren des Multiprozessorsystems fast immer die (gleiche) Struktur der im Rahmen dieser Arbeit behandelten Anwendungsbeispiele besitzen. Nach Durchlaufen einer Initialisierungsphase, in der nur auf Daten im "eigenen" Speicherbereich zugegriffen wird, erfolgt eine Synchronisation aller Prozessoren ("Warten, bis alle Prozessoren ihre Initialisierungsphase durchlaufen haben"), bevor der eigentliche Verarbeitungsteil (mit Kommunikation mit anderen Prozessoren) beginnt. Eine solche Programmstruktur und eine Programmierungsumgebung, die einen derartigen Programmaufbau für Parallelprogramme regelrecht erzwingt, sind deshalb sinnvoll, da bereits mit dem ersten Informationsaustausch zwischen verschiedenen Prozessoren Annahmen über einen aus der Initialisierung des fremden Prozessors entwickelten Programmzustand getroffen werden können und nicht von völlig zufällig gestalteter Fremdinformation ausgegangen werden muß.

Wenn die Steuerung von Prozessen in einem Parallelprogramm über einen dem Benutzer in seiner Struktur verborgenen Rahmen auf Hauptprogrammebene erfolgt und die Benutzerprogramme für einzelne Prozessoren als vom Rahmen aufgerufene Unterprogramme zu deklarieren sind (vgl. Anmerkungen zu Beginn von Kap. 5.1), ergibt sich damit fast zwangsläufig folgender Aufbau:

1. In einer Routine "Initialisierung", mit deren Aufruf vom Rahmen nur die prozessorspezifische Kennung an das Benutzerprogramm gemeldet wird, sind die zum Prozessor gehörigen Daten, die für andere Prozessoren erreichbar sein sollen, auf einen Initialzustand zu bringen.

Es läßt sich denken, daß bei besonderer Kennzeichnung dieser Daten ein Interpreter abprüft, ob in der Initialisierungsphase an jede zu diesem Bereich gehörige Variable ein Wert zugewiesen wurde.

In den Programmierungsumgebungen für EGPA-artige Rechnerarchitekturen entfällt der Aufruf der Initialisierungsroutine für den Kontrollprozessor, da auf dessen Speicherbereich keiner der übrigen Prozessoren Zugriff hat.

2. In einer Routine "Verarbeitung", mit deren Aufruf vom Rahmen Kennungen erreichbarer Nachbarprozessoren an das Benutzerprogramm gemeldet werden, erfolgt der eigentliche Programmlauf, der Teile zur Kommunikation mit anderen Prozessoren einschließt. Die Routine "Verarbeitung" wird vom Rahmen erst dann aufgerufen, wenn sämtliche Prozessoren des Systems ihren Initialisierungsteil durchlaufen haben.

Auch Parallelprogramme, die mit Hilfe von Auftragssystemen, wie sie der Philosophie des Makro-Datenflusses [Knei82] zugrunde liegen, formuliert werden, bilden diese Struktur nach. Nach Vergabe eines Auftrages zur Initialisierung an alle Prozessoren wird auf dessen Abschluß gewartet, bevor neue Aufträge zur Bearbeitung an andere Prozessoren delegiert werden. Die in [Knei82] beschriebene Pilotimplementierung eines Makro-Datenfluß-Systems für das EGPA-System, die aus der Sicht der elementaren Programmierungsumgebung des Kap. 5.1.1 als Anwenderprogramm gesehen wird, folgt exakt diesem Aufbau.

6.2 Sprachmittel, Einbettung in Organisationsformen für Multiprozessorsysteme

Kennzeichen der "E/A-orientierten" Implementierungsmethode ist, daß Informationen zwischen verschiedenen Programm-Modulen bzw. Prozessoren eines Multiprozessorsystems nur über Kommunikationsprozeduren ausgetauscht werden. Diese werden so implementiert, daß Informationen mittels Wertzuweisungen in Puffer eingeschrieben bzw. aus Puffern ausgelesen werden.

In Systemen, bei denen Prozessoren auf einen gemeinsam benutzbaren Speicherbereich zugreifen können, werden die Puffer in diesem Kommunikationsspeicher angelegt, in botschaftsorientierten Systemen werden die Puffer im Privatspeicher des Prozessors angelegt und bei Bedarf als Bestandteil einer Botschaft an den Kommunikationspartner übertragen. Je nachdem, wie häufig in letzterem Fall Informationen zwischen Prozessoren übertragen werden, ist von einem Prozessor aus eine aktuelle Information über den Zustand eines anderen Prozessors möglich. Im folgenden sollen die Bereiche, in denen anderen Prozessoren zugängliche Daten abgelegt werden, als Kommunikationsspeicher bezeichnet werden.

Wenn eine Form der Programmierung des Multiprozessorsystems vorgenommen wird, in der auf verschiedenen Prozessoren die gleichen Programme ablaufen und diese sich wie hier beschrieben mittels Kommunikationsprozeduren über Puffer "unterhalten", ergeben sich zur Realisierung der Kommunikation zwei Klassen von Operationen:

1. Prozessorbezogene Operationen, die sich auf einen bzw. eine Menge direkt erreichbarer Nachbarprozessoren beziehen und
2. pufferbezogene Operationen, die sich auf festgelegte Bereiche im Kommunikationsspeicher eines bestimmten Prozessors beziehen, d. h. im allgemeinen erst nach Ausführung mindestens einer Operation der ersten Klasse erfolgen können.

Neben einer prozessorbezogenen Operation

PSWITCH (<Prozessorkennung>)

zum Umschalten auf den Kommunikationsspeicher des durch <Prozessorkennung> bezeichneten Prozessors lassen sich prozessorbezogene Operationen denken wie

"Warte auf einen bestimmten Prozessor",

"Warte auf einen beliebigen erreichbaren Prozessor",

"Warte auf alle erreichbaren Prozessoren",

welche nachfolgend noch genauer spezifiziert werden.

Nach Umschalten auf den Kommunikationsspeicher eines bestimmten Prozessors lassen sich pufferbezogene Operationen ausführen. Dabei werden verschiedene Puffer im Kommunikationsspeicher eines Prozessors durch verschiedene Kanalnummern unterschieden.⁵⁷⁾ Eine Abbildung von Kanalnummern auf verschiedene Puffer kann realisiert werden, indem die Kanalnummer als Index in einem Feld verwendet wird, dessen Elemente Referenzen auf die jeweiligen Puffer bzw. deren erstes Element sind.

Die Puffer mit den sie beschreibenden Größen können je nach den verfügbaren Sprachmitteln in Form eigener Segmente oder Strukturen deklariert werden.

57) Wenn pro Prozessor nur ein Puffer für Nachrichten an andere Prozessoren erforderlich ist, kann auf Kanalnummern und damit auf eine Stufe zur Adreßumsetzung beim Pufferzugriff verzichtet werden. Derartig gebildete Kommunikationsprozeduren können als Standard-E/A-Operationen angesehen werden (vgl. UNIX-E/A-Operationen "printf", "scanf" zu "fprintf" und "fscanf").

Für die im Rahmen dieser Arbeit implementierten Anwendungen wurde von folgender Pufferstruktur ausgegangen:

GROESSE	Zahl der Elemente, die der Puffer speichern kann
READSTATE	Zählt, wie oft der Puffer zum Lesen eröffnet und wie oft nach Lesen des letzten Elementes geschlossen wurde
WRITESTATE	Zählt, wie oft der Puffer zum Schreiben eröffnet und wie oft wieder geschlossen wurde
READANZ	Zahl der aus dem Puffer gelesenen Elemente
WRITEANZ	Zahl der in den Puffer geschriebenen Elemente
ELEMENT	Feld zur Aufnahme der vom Puffer zu speichernden Elemente (die einzige Stelle, an der der Typ des Puffers festgelegt wird)

Dazu kommen auf Modulebene als für fremde Prozessoren nicht zugängliche Information hinzu:

READPTR	Position im Puffer, von der das nächste Element zu lesen ist
WRITEPTR	Position im Puffer, auf die das nächste Element zu schreiben ist
OPENSTATE	Zur Prüfung, ob eine Kommunikationsprozedur zulässig ist (z. B. mit den Werten 0: "Puffer geschlossen", -1: "Puffer zum Lesen eröffnet" und 1: "Puffer zum Schreiben eröffnet")
CLOSESTATE	zum Festhalten des letzten Wertes von READSTATE bzw. WRITESTATE (bei Schliesse-Operationen aktualisiert)

Diese Größen dienen mit dazu, die auf den Puffer auszuführenden Operationen reihenfolgerichtig zu synchronisieren.

Damit können typische Kommunikationsprozeduren auf Puffern innerhalb eines Kommunikationsmoduls einer parallelen Programmierungsumgebung in der nachfolgend beschriebenen Art realisiert werden.

Funktion: QU := PDECL (<Kanal>, <Größe>);

Bedeutung: Der durch <Kanal> beschriebene Puffer wird mit der durch <Größe> angegebenen Kapazität initialisiert. Falls die Routine korrekt ausgeführt worden ist, wird als Ergebnis QU (wie auch für die nachfolgenden Operationen) die "Quittung "korrekt" zurückgegeben, sonst eine Fehlerquittung, die die Ursache der Nichtausführbarkeit der Operation anzeigt. Mögliche Fehler sind "Kanalnummer nicht zulässig" oder "Wert von <Größe> nicht zulässig".

Realisierung: Im durch <Kanal> bezeichneten Puffer werden die den Puffer beschreibenden Größen auf ihren Initialwert gebracht (GROESSE auf den im entsprechenden Parameter mitgegebenen Wert). Aus Konsistenzgründen muß gefordert werden, daß diese Operationen bezgl. der von mehr als einem Prozeß erreichbaren Pufferdaten nur einmal zum Programmbeginn von dem Prozeß durchgeführt werden dürfen, der anschließend den Puffer beschreibt.

Funktion: QU := POPENR (<Kanal>);

Bedeutung: Der nach Ausführung des entsprechenden PDECL-Aufrufes durch <Kanal> spezifizierte Puffer wird zum Lesen (Read) eröffnet.

Realisierung: Es wird so lange gewartet, bis der durch <Kanal> spezifizierte Puffer (vom fremden Prozessor) zum Schreiben eröffnet wurde (WRITESTATE > READSTATE und WRITESTATE > CLOSESTATE).

Anschließend wird der Lesezeiger READPTR so gesetzt, daß er auf das erste Element zeigt, die Größe READSTATE erhöht und die Größe OPENSTATE auf den Wert "Puffer zum Lesen eröffnet" gebracht.

Funktion: QU := POPENW (<Kanal>);

Bedeutung: Der nach Ausführung des entsprechenden PDECL-Aufrufes durch <Kanal> spezifizierte Puffer wird zum Schreiben (Write) eröffnet.

Realisierung: Es wird so lange gewartet, bis für den durch <Kanal> spezifizierten Puffer die Größe READSTATE den aktuellen Wert von WRITESTATE erreicht hat (Vermeidung des Überschreibens noch nicht gelesener Information). Anschließend wird der Schreibzeiger WRITEPTR so gesetzt, daß er auf das erste Element zeigt, die Größe WRITESTATE erhöht und die Größe OPENSTATE auf den Wert "Puffer zum Schreiben eröffnet" gebracht.

Funktion: QU := PCLOSE (<Kanal>);

Bedeutung: Der durch <Kanal> beschriebene Puffer wird geschlossen. Dies hat zur Folge, daß danach folgende diesen Puffer betreffende Schreibaufrufe nicht mehr ausgeführt werden und die Quittung "Datenende" als Funktionsergebnis zurückgegeben wird.

Realisierung: Bei dem durch <Kanal> beschriebenen Puffer wird die Größe OPENSTATE auf den Wert "Puffer geschlossen" gesetzt, WRITESTATE erhöht und CLOSESTATE an den neuen Wert von WRITESTATE angeglichen.

Für die eigentlichen Kommunikationsprozeduren auf Puffern lassen sich wie bei der Datei-E/A zwei Varianten vereinbaren, unformatierte E/A und formatierte E/A. Bei der unformatierten E/A, die in Anlehnung an die UNIX-E/A-Primitiva "read" und "write" definiert ist, wird eine angebbare Zahl von Elementen des Typs des angesprochenen Puffers übertragen, bei der formatierten E/A eine Menge von Daten, die in einem vom Benutzer spezifizierten Format vorliegen. Die formatierte E/A kann wie über die UNIX-Primitiva "sprintf" und "sscanf" realisiert werden. Beim Schreiben wird das benutzerspezifizierte Datenformat in ein Bytefeld "geschrieben", welches anschließend unformatiert übertragen wird; beim Lesen wird ein Bytefeld empfangen, aus dem anschließend das benutzerspezifizierte Datenformat entnommen und entsprechend interpretiert wird.

Funktion: QU := PREAD (<Kanal>, <Info>, <Menge>);

Bedeutung: Aus dem durch <Kanal> bestimmten Puffer wird die in <Menge> angegebene Zahl von Informationseinheiten (unformatiert) ausgelesen und an der unter <Info> angegebenen Stelle abgespeichert. Bei korrektem Verlauf wird zurückgemeldet, wieviele Elemente aus dem Puffer heraus übertragen wurden.

Realisierung: Es wird z. B. mit der Funktion PWAITR (vgl. unten) so lange gewartet, bis die zu lesende Information in der angegebenen Menge vorhanden ist. Ist dies der Fall, wird die Information entnommen und der Lesezeiger READPTR aktualisiert. Wird "Datenende" festgestellt (kein Element mehr im Puffer enthalten und WRITESTATE > READSTATE), werden die Größen READSTATE und CLOSESTATE an WRITESTATE angeglichen und die Größe OPENSTATE auf den Wert "Puffer geschlossen" gesetzt. In diesem Fall wird als Quittung (wie auch für nachfolgende Leseaufrufe) "Datenende" zurückgemeldet.

- Funktion: QU := PWRITE (<Kanal>, <Info>, <Menge>);
- Bedeutung: In den durch <Kanal> bestimmten Puffer wird die in <Menge> angegebene Zahl von Informationseinheiten von der durch <Info> bezeichneten Stelle ab eingetragen.
- Realisierung: Es wird z. B. mit der Funktion PWAITW (vgl. unten) so lange gewartet, bis im angesprochenen Puffer ausreichend Platz zur Aufnahme der einzutragenden Information vorhanden ist. Ist dies der Fall, wird die Information eingetragen und der Schreibzeiger WRITEPTR aktualisiert.

Die Funktionen

QU := PINPUT(<Kanal>, <Format>, <Info>); und

QU := POUTPUT (<Kanal>, <Format>, <Info>);

zur formatierten E/A lassen sich nach den oben beschriebenen Umformungen auf die Funktionen PREAD und PWRITE abbilden.

Die in Kap. 3 beschriebenen parallelen Verarbeitungsstrategien für ordnungserhaltende Programmsysteme enthalten neben Aufrufen von Kommunikationsprozeduren zur Gewinnung von Informationen über Zustände von Programmen auf "fremden" Prozessoren direkt Aufrufe wie "Warte auf nächstes Ergebnis" u. ä. In diesem Fall ist nicht der Inhalt der Information von Bedeutung, sondern allein die Tatsache, daß die Information vorhanden ist bzw. abgelegt werden kann, wobei mit dem Aufruf noch keine Informationsübertragung stattfindet. Deshalb erscheint es sinnvoll, neben den Kommunikationsprozeduren auch die von den Kommunikationsprozeduren verwendeten Synchronisationsprozeduren auf einer eigenen Modulebene (vgl. Abb. 5.5) zur Verfügung zu stellen. Diese lassen sich etwa in der nachfolgend beschriebenen Weise definieren.

Funktion: QU := PWAITR (<Kanal>, <Menge>)

Bedeutung: Es wird gewartet, bis der durch <Kanal> bezeichnete Puffer mindestens die in <Menge> angegebene Zahl von Informationseinheiten enthält.
Ein nicht korrektes Funktionsergebnis zeigt in diesem Fall einen Endezustand oder eine Systemverklemmung an (vgl. Kap. 5.3).

Realisierung: Es wird über einen der in Kap. 5.2 skizzierten Synchronisationsmechanismen so lange gewartet, bis für den durch <Kanal> bezeichneten Puffer die Größen READANZ, WRITEANZ und <Menge> der Beziehung

$$\text{WRITEANZ} \geq \text{READANZ} + \text{<Menge>}$$

genügen. In botschaftsorientierten Systemen wird so lange eine neue Botschaft angefordert (und anschließend entsprechend interpretiert), bis diese Beziehung erfüllt ist.

Funktion: QU := PWAITW (<Kanal>, <Menge>);

Bedeutung: Es wird gewartet, bis der durch <Kanal> bezeichnete Puffer Platz zur Aufnahme der in <Menge> angegebenen Zahl von Informationseinheiten hat.
Wie bei PWAITR zeigt auch hier ein nicht korrektes Funktionsergebnis einen Endezustand oder eine Systemverklemmung an.

Realisierung: Es wird über einen der in Kap. 5.2 skizzierten Synchronisationsmechanismen so lange gewartet, bis für den durch <Kanal> bezeichneten Puffer die Größen GROESSE, READANZ, WRITEANZ und <Menge> der Beziehung

$$\text{READANZ} \geq \text{WRITEANZ} - \text{GROESSE} + \text{<Menge>}$$

genügen. In botschaftsorientierten Systemen wird so lange eine neue Botschaft angefordert (und anschließend entsprechend interpretiert), bis diese Beziehung erfüllt ist.

Funktion: QU := PSIGNR (<Kanal>, <Menge>);

Bedeutung: Es wird (in der Regel als Ergebnis einer WRITE-Operation) angezeigt, daß nachfolgend aus dem durch <Kanal> bezeichneten Puffer wieder mindestens die durch <Menge> angezeigte Zahl von Informationseinheiten entnommen werden kann. Wenn bezgl. des angesprochenen Puffers ein Zustand vorliegt, in dem die Ausführung der Operation nicht sinnvoll ist ("Datenende erreicht" oder "Systemverklemmung"), wird dies an den Benutzer über eine entsprechende Quittung QU weitergegeben.

Realisierung: Bei möglicher Ausführung der Funktion wird die Größe WRITEANZ des durch <Kanal> bezeichneten Puffers um den in <Menge> angegebenen Wert erhöht und ein evtl. über Interrupt blockierter Prozeß, der danach wieder lauffähig ist, befreit. In botschaftsorientierten Systemen wird ggf. eine Botschaft abgesetzt, mit der andere Prozesse des Systems über die Zustandsänderung unterrichtet werden.

Funktion: QU := PSIGNW (<Kanal>, <Menge>);

Bedeutung: Es wird (in der Regel als Ergebnis einer READ-Operation) angezeigt, daß nachfolgend in den durch <Kanal> bezeichneten Puffer wieder mindestens die durch <Menge> angezeigte Zahl von Informationseinheiten eingetragen werden kann. Wenn bezgl. des angesprochenen Puffers ein Zustand vorliegt, in dem die Ausführung der Operation nicht sinnvoll ist ("Datenende erreicht" oder

"Systemverklemmung"), wird dies an den Benutzer über eine entsprechende Quittung QU weitergegeben.

Realisierung: Bei möglicher Ausführung der Funktion wird die Größe READANZ des durch <Kanal> bezeichneten Puffers um den in <Menge> angegebenen Wert erhöht und ein evtl. über Interrupt blockierter Prozeß, der danach wieder lauffähig ist, befreit. In botschaftsorientierten Systemen wird ggf. eine Botschaft abgesetzt, mit der andere Prozesse des Systems über die Zustandsänderung unterrichtet werden.

Mit Hilfe dieses Satzes von Funktionen ist es bereits möglich, alle parallelen Verarbeitungsstrategien zu formulieren, die deterministisch ablaufen, wie z. B. die in Kap. 3.2.2 beschriebene Round-Robin-Heuristik zur parallelen Verarbeitung ordnungserhaltender Programmsysteme. Wie die Anmerkungen zu den einzelnen Funktionen zeigen, ist es dabei vom Implementierungsstandpunkt gesehen ohne Bedeutung, ob im zugrundeliegenden Multiprozessorsystem Kommunikation zwischen Prozessoren über gemeinsam benutzte Speicherbereiche oder über Botschaften mit festgelegtem Absender und festgelegtem Empfänger realisiert ist (vgl. jedoch Fußnote 6 in Kap. 1).

Die vorgestellten Funktionen, die einen synchronisierten Informationsaustausch⁵⁸⁾ zwischen den Prozessen eines Parallelprogramms definieren, lassen sich ohne Schwierigkeiten auf nicht synchronisierte Parallelprogramme erweitern.

58) Unter einem synchronisierten Informationsaustausch soll verstanden werden, daß keine noch nicht gelesene Information zwischen Prozessen durch Überschreiben verloren geht und daß die Leseroutine erst dann verlassen wird, wenn die zu lesende Information verfügbar war und übertragen wurde. Ein nicht synchronisierter Informationsaustausch zwischen Prozessoren wird z. B. bei Parallelimplementierungen zur Lösung von Aufgaben der numerischen Mathematik über iterative Näherungsverfahren verwendet.

Eine Möglichkeit wäre, in den Kommunikationsprozeduren die Zahl zu übertragender Informationseinheiten durch einen negativen Wert anzugeben. In diesem Fall würde in Kommunikationsprozeduren nicht auf die Erfüllung von Bedingungen zum Programmfortschritt gewartet. Dies hätte zur Folge, daß noch nicht gelesene Information überschrieben werden kann oder bei Leseaufrufen keine "neuen" Informationen entnommen werden. Derartige Ereignisse könnten über entsprechende Quittungen an den Benutzer weitergegeben werden.

Zur Realisierung speicherorientierter Heuristiken (vgl. Kap. 3.2.4) auf der Basis von Kommunikationsprozeduren muß der vorgestellte Satz von Operationen noch um zwei Funktionen erweitert werden, die über aktuelle Belegungszustände bestimmter Puffer informieren, z. B.

```
QU := PBELEGT (<Kanal>;
```

zur Ermittlung, wieviele Elemente momentan noch aus dem durch <Kanal> bezeichneten Puffer gelesen werden können und

```
QU := PFREI (<Kanal>;
```

zur Ermittlung, wieviele Elemente momentan noch in den durch <Kanal> bezeichneten Puffer eingetragen werden können, ohne daß mit einem Blockieren des Prozesses gerechnet werden muß. Im Funktionsergebnis möge der jeweilige Zahlenwert zurückgegeben werden bzw. eine (negative) Fehlerquittung, wenn ein Endezustand mit leerem Puffer oder eine Systemverklemmung vorliegt.⁵⁹⁾

Hier erweisen sich Systeme mit einer Kommunikation zwischen Prozessoren über gemeinsame Speicherbereiche gegenüber botschaftsorientierten Systemen überlegen. Informationen sind aktuell und einfach über Wertzuweisung erhalten. In botschaftsorientierten Systemen wird entweder viel Zeit zur Ermittlung eines aktuellen Zustands durch Empfang noch ausstehender

59) Bei "zulässigem" Zustand des jeweils angesprochenen Puffers wird als Ergebnis von PBELEGT das Ergebnis von WRITEANZ-READANZ, bei PFREI das Ergebnis von GROESSE-(WRITEANZ-READANZ) zurückgegeben.

Botschaften benötigt, oder ein bezgl. eines Puffers ermittelter Zustand stimmt nicht mit dem aktuell gültigen Zustand überein.⁶⁰⁾

Ein wenig "besser" ist die Situation bei der Realisierung der zustandsorientierten Heuristik des Kap. 3.2.3. Hier ist es bedeutungslos, ob auf das Ergebnis, welches als nächstes entsteht, über Abfragen von Zustandsgrößen in einem allen Prozessoren zugänglichen Speicherbereich oder über einen Botschaftsmechanismus erfolgt, der ein Warten auf die Botschaft eines beliebigen Absenders gestattet.

Für derartige Fälle erweist es sich als zweckmäßig, den Satz bislang vorgestellter Operationen um prozessorbezogene Operationen wie

QU := WAITNEXT (<Kanal>);

- "Warte, bis die nächste Informationseinheit bezgl. <Kanal> auf mindestens einem der erreichbaren Nachbarprozessoren vorliegt", oder

QU := WAITALL (<Kanal>);

- "Warte, bis die nächste Informationseinheit bezgl. <Kanal> auf allen erreichbaren Nachbarprozessoren vorliegt"

zu ergänzen.

In die gleiche Kategorie fällt die im Zusammenhang mit der in Abb. 3.15 dargestellten Rechnerarchitektur mit global zugänglichem Eingabepuffer erläuterte dezentrale Parallelverarbeitung ordnungserhaltender Programmsysteme. Hier ist es vom Standpunkt der Implementierung her ohne Bedeutung, ob das Vorliegen eines

60) In einer Implementierung unter Verwendung der in Kap. 5.1.2 skizzierten UNIX-Umgebung wurde zur Vermeidung eines häufigen Sendens von Botschaften erst dann eine neue Botschaft mit aktualisierten Zustandswerten ausgesendet, wenn der laufende Prozeß blockiert zu werden drohte. Dadurch ließ sich aus ermittelten Belegungswerten bezgl. eines Puffers überhaupt kein Bild mehr über den tatsächlich aktuell gültigen Zustand gewinnen.

neu eingelesenen Datenblocks zur Weiterverarbeitung einer Menge von Prozessoren über Zustandsgrößen in einem allgemein zugänglichen Speicher oder über eine allgemein zugängliche Botschaft übermittelt wird.⁶¹⁾

Damit lassen sich die parallelen Verarbeitungsstrategien für ordnungserhaltende Programmsysteme bezgl. der Anforderungen an die Realisierung der Prozessorkommunikation im Multiprozessor-Betriebssystem klassifizieren.

Zur Realisierung von Heuristiken vom Typ der Round-Robin-Heuristik in Parallelprogrammen (Parallelprogrammen mit deterministischem Programmablauf, Parallelprogrammen mit statischer Dekomposition bezgl. der Zuordnung von Teilaufgaben zu den verfügbaren Prozessoren des Systems) reichen Mechanismen zur Kommunikation zwischen Prozessoren aus, bei denen Sender und Empfänger von Informationen immer eindeutig bestimmt sind.

Zur effizienten Realisierung zustandsorientierter Heuristiken für Parallelverarbeitung ist es zumindest notwendig, zusätzlich einen Mechanismus zur Prozessorkommunikation zur Verfügung zu haben, der ein Senden von Information an beliebige Empfänger bzw. ein Empfangen von Information von einem beliebigen Absender gestattet.

Zur effizienten Realisierung speicherorientierter Heuristiken für Parallelverarbeitung (von Parallelprogrammen mit dynamischer von Zuständen fremder Prozessoren während des Programmablaufs abhängiger Dekomposition bezgl. der Zuordnung von Teilaufgaben zu den verfügbaren Prozessoren des Systems) sind darüberhinaus Mechanismen erforderlich, die "schnell" und aktuell eine Ermittlung von Zuständen fremder Prozessoren ermöglichen. Gerade in letzterem Fall scheint eine Kommunikation zwischen Prozessoren über Botschaften einer Kommunikation zwischen Prozessoren über gemeinsam genutzte Speicherbereiche unterlegen zu sein.

61) In [Iss83] werden diese beiden Formen der Kommunikation als "Kommunikation über einen öffentlichen Eingang" bzw. "Kommunikation über einen öffentlichen Ausgang" bezeichnet.

Auch zur Realisierung von Implementierungen unter Ausnutzung einer Nachbarhilfe zwischen Prozessoren (vgl. etwa Ausführungen zum Ende von Kap. 4.2.2) gelten die Anmerkungen bezgl. der Effizienz in der Anwendung der verschiedenen Kommunikationsmechanismen. Auch hier ist eine aktuelle und "schnell" erreichbare Information über den Zustand fremder Prozessoren notwendig.

Eine mit Kommunikationsprozeduren realisierbare Implementierungsform, bei der nur die Position im Puffer von Bedeutung ist, von der eine Information gelesen bzw. auf die eine Information geschrieben wird, nicht jedoch die Reihenfolge, in der Kommunikationsprozeduren bezgl. bestimmter Positionen auszuführen sind, läßt sich in der Form definieren, daß mit der Kommunikationsprozedur die betreffende Pufferposition angegeben wird. Lese- und Schreibzeiger sind in diesem Fall ohne Bedeutung. Ein Arbeitsende wird in diesem Fall durch Information über den Belegungszustand des bearbeiteten Puffers ermittelt.

Diese Art, eine Nachbarhilfe zwischen Prozessoren zu implementieren, ist jedoch nur effizient, wenn (wie im Programmbeispiel des Kap. 4.2.2)

1. Berechnungen weitgehend prozessorlokal möglich sind,
2. Berechnungsprozeduren signifikant längere Zeit in Anspruch nehmen als Kommunikationsprozeduren und/oder
3. mit Kommunikationsprozeduren nur sehr geringe Mengen an Information (z. B. einzelne Zahlenelemente) zu übertragen sind.

In Programmsystemen, in denen der Aufwand für Kommunikation einen im Verhältnis zur Rechenzeit nicht vernachlässigbaren Anteil ausmacht, können die Kommunikationsprozeduren zum Engpaß bei der Erreichung eines möglichst hohen Geschwindigkeitsgewinns werden. Dies mag der Punkt sein, an dem "direkten" Methoden zur parallelen Implementierung letztendlich der Vorzug gegeben werden muß und eine E/A-orientierte parallele Implementierung mehr als Übergangslösung (Testhilfsmittel) bei der Programmentwicklung angesehen werden sollte.

7 Zusammenfassung und Ausblick

Ordnungserhaltende Programmsysteme lassen sich ohne Änderungen an (sequentiellen) Benutzeralgorithmen auf einem Multiprozessorsystem implementieren, wenn

1. sie in nach Methoden des Composite Design konstruierte Teilaufgaben (Phasen zur Transformation der zu verarbeitenden Datenmenge) partitioniert sind
2. die entstandenen Teilaufgaben als Prozeduren implementiert werden und
3. der Informationsaustausch zwischen Teilaufgaben über (eigens hierzu erstellte) Kommunikationsprozeduren erfolgt.

Dann ist es zumindest möglich, durch Anpassung der Kommunikationsprozeduren an die vom Betriebssystem zur Verfügung gestellten Mechanismen zur Prozessorkommunikation einzelne Teilaufgaben einzelnen Prozessoren zuzuordnen. Im Ergebnis ergibt sich eine pipelineartige Verarbeitung (Kap. 2.2 und 3.1).

Wenn es gelingt, die Menge der zu verarbeitenden Daten in eine bezgl. des Programmsystems informationskonsistente Folge von Datenblöcken zu partitionieren, ergibt sich eine zweite Dimension von Teilaufgaben. Eine einzelne Teilaufgabe besteht dann nicht mehr darin, den gesamten Datenbestand (gemäß Berechnungsvorschrift) zu transformieren, sondern lediglich noch darin, einen einzelnen Datenblock zu transformieren. Für ein ordnungserhaltendes Programmsystem kann die Abhängigkeit der Teilaufgaben untereinander durch einen Graphen beschrieben werden, der (bei Zuordnung der Teilaufgaben zu Prozessoren eines Multiprozessorsystems) nach der Wellenfront-Methode abgearbeitet werden kann (Kap. 2.3 und 2.4).

Weitere Möglichkeiten der parallelen Verarbeitung ergeben sich, wenn in diesem Graphen dargestellte Abhängigkeiten zwischen den Teilaufgaben entweder real nicht existieren oder durch Ändern von Teilaufgaben oder zu verarbeitenden Datenblöcken abgebaut werden können. Je mehr Teilaufgaben auf diese Weise entflochten werden

können, desto bessere Möglichkeiten bestehen, in Multiprozessor-systemen zu zufriedenstellenden Parallel-Implementierungen zu gelangen (Kap. 2.4, 3.2, 3.3). Neben der durch den Kontroll- und Datenfluß bedingten Einschränkung, daß aufeinanderfolgende Teilaufgaben, die den gleichen Datenblock transformieren, nicht entflochten werden dürfen, können speziell in ordnungserhaltenden Programmsystemen die Teilaufgaben "Eingabe der ursprünglichen Daten" und "Ausgabe der schließlich erzielten Ergebnisse" nicht entflochten werden. ("Alles parallel zu machen, geht nicht!")

Wenn die Zahl der Teilaufgaben eines Programmsystems größer als die Zahl der verfügbaren Verarbeitungseinheiten (Prozessoren) im Multiprozessorsystem ist, ist die Zuordnung von Aufgaben zu Prozessoren im Hinblick auf die Erzielung einer Lösung mit minimaler Gesamtausführungszeit häufig nur über Heuristiken in vertretbarem Aufwand durchführbar. Meist liegt dies daran, daß das Zuordnungsproblem - wie das allgemeine Multiprozessor-Scheduling-Problem - NP-vollständig ist oder - wie für die untersuchten ordnungserhaltenden Programmsysteme - keine optimalen Lösungen bekannt sind. Je "leichter" Information über den Bearbeitungszustand von Aufgaben auf fremden Prozessoren verfügbar gemacht werden können, desto bessere Möglichkeiten ergeben sich, mit Heuristiken zu fast optimalen Zuordnungen zu gelangen (Kap. 3.2, 4.1.2, 6.2).

Als ein Ergebnis der durchgeführten Experimente kann festgehalten werden, daß solche Informationen in der notwendigen Geschwindigkeit und Genauigkeit nur in speichergekoppelten Systemen wie EGPA erhalten werden können. Das Verhältnis zwischen den Zugriffen auf den eigenen Speicher und Zugriffen auf Speicher fremder Prozessoren kann also (vgl. [Horn84]) kein Kriterium sein, die Notwendigkeit einer Speicherkopplung zwischen Prozessoren in Frage zu stellen. Ohne die mit der Speicherkopplung zwischen Prozessoren erreichbaren Informationen wird man letztlich immer auf parallele Verarbeitungsstrategien festgelegt sein, mit denen es nicht möglich ist, Einflüsse von Multiprozessor-Architektur und Multiprozessor-Betriebssystem (z. B. unterschiedlich schnelle

Prozessoren, Bevorzugung von Prozessoren bei der Bedienung durch das Betriebssystem und dgl.) über die Zuordnung von Aufgaben zu Prozessoren auszugleichen.

Die aus der Systematik zur Parallelisierung entstandenen parallelen Programmstrukturen erlauben es, spezifische Probleme paralleler Verarbeitung wie das Zuordnungsproblem ohne Änderungen von Benutzeralgorithmen zu untersuchen. Im Hinblick auf die Programmierbarkeit paralleler Lösungen lassen sich die wesentlichen Vorteile des durchgeführten Weges in folgenden zwei Punkten zusammenfassen:

1. Entstandene Parallelprogramme lassen sich ohne Änderung an Benutzeralgorithmen in verschiedenen Mono- und Multiprozessor-systemen mit verschiedenen Verarbeitungsstrategien ausführen. Abgesehen von der Gestaltung sequentieller und paralleler Verarbeitungsstrategien (die in Abhängigkeit von der Struktur des Systems und der zu verarbeitenden Aufgabenmenge zu formulieren ist) und Aspekten der Programmiersprache reicht es im allgemeinen aus, beim Übergang auf ein "neues" System die Kommunikationsmodule auszuwechseln. Dies gilt (bei korrekter Implementierung der Kommunikation) unabhängig davon, ob die Kommunikation botschaftsorientiert oder prozedurorientiert ist (Kap. 5.1, 5.2). Zu beachten ist jedoch stets, daß unterschiedliche Kommunikations- und Synchronisationstechniken im Parallelprogramm zu unterschiedlichem Prozeßverhalten und damit zu unterschiedlich erreichbaren Geschwindigkeitsgewinnen führen.
2. Die Entwicklung fehlerfreier Parallelprogramme wird beschleunigt, da Programmtests entweder im sequentiellen Programm oder im Parallelprogramm auf einer "Systemebene" (Kommunikationsprozeduren einer parallelen Programmierungsumgebung) durchgeführt werden können. Wenn die Kommunikation zwischen Prozessoren ausschließlich über Prozeduren einer parallelen Programmierungsumgebung erfolgt, kann darüberhinaus "systemseitig" die Verklemmungsfreiheit der Parallelprogramme geprüft werden (Kap. 5.3 und 5.4).

Diese Art, Programme zu parallelisieren, eignet sich vor allem für "große" ordnungserhaltende Programmsysteme. Warum?

1. Aufgrund von Umfang, Komplexität und "Sequentialität" der Einzelprogramme wird im allgemeinen vor Änderungen am Benutzeralgorithmus zum Zweck der Parallelisierung zurückgeschreckt. Es wurde z. B. nie ernsthaft erwogen, originäre parallele Algorithmen für die Teilaufgaben der Textformatierung (vgl. Kap. 4.1) zu entwickeln. (Inzwischen ist der Verfasser der Ansicht, daß dies auch gar nicht geht!)
2. Aufgrund von Umfang und Komplexität werden derartige Programmsysteme bereits auf der Seite des sequentiellen Programms in Teilaufgaben (Phasen) partitioniert (vgl. etwa die Phasen eines Compilers). Damit ist ein Teil der Parallelisierung nach der hier dargestellten Methode bereits erfolgt.
3. Wegen der Ordnungserhaltung zwischen Ein- und Ausgaben kann die Korrektheit des sequentiellen Programms schon weitgehend durch Protokollierung der Abfolge von Ein- und Ausgaben getestet werden. Zu solchen Tests ist es bereits im sequentiellen Programmsystem notwendig, eigene Kommunikationsprozeduren zu formulieren. Liegt es da nicht nahe, diese Arbeit in die Gestaltung paralleler Programme mit einzubeziehen?

Der Ansatz hat jedoch auch Bedeutung für beliebige Klassen zu parallelisierender Programme. Mit der Konstruktion einer sequentiellen deterministisch ablaufenden Verarbeitungsstrategie wird nämlich eine Ordnungserhaltung zwischen Ein- und Ausgaben festgelegt! Die Folge ist: Der Ansatz zur Parallelisierung kann auch für andere Klassen von Programmsystemen zumindest als Testgrundlage bei der Programmentwicklung angewendet werden (vgl. Kap. 4.2). Ob damit allerdings immer zufriedenstellende Lösungen erzielt werden, ist stark von der jeweiligen Anwendung abhängig. Eine große Rolle spielt dabei das Verhältnis zwischen (prozessor-autonomer) Rechenzeit und Zeit zur Kommunikation mit anderen Prozessoren. Die Notwendigkeit, für bestimmte Anwendungen zur Erreichung eines zufriedenstellenden Geschwindigkeitsgewinns

durch Parallelverarbeitung völlig neue Algorithmen zu entwickeln (z. B. zur Parallelisierung der Polynomberechnung mit Hilfe des Hornerschemas), kann dadurch nicht in Frage gestellt werden.

Warum wurde dennoch in der beschriebenen Art bei der Parallelisierung vorgegangen? Warum wurden auch bei der Parallelisierung "einfacher" Aufgabenstellungen Kommunikationsanweisungen eingeführt, obwohl das verfügbare EGPA-System mit der Speicherkopplung die Möglichkeit bot, "direkte" Lösungen zu implementieren?

Ein solcher Ansatz erschien dem Verfasser als einzige Möglichkeit, beim gegenwärtigen Stand von Hard- und Softwaretechnologie Parallelprogramme "testbar" zu machen. Ein solcher Ansatz entstand nicht nur aus eigenen Erfahrungen mit der Implementierung von Parallelprogrammen auch in (simulierten) größeren Multiprozessorsystemen als der EGPA-Pilotpyramide (vgl. [Rat83]). Er wurde hauptsächlich aus der über Jahre hinweg durchgeführten Benutzerberatung motiviert. Weit über 90 % der Beratungen bestanden in der Beantwortung der Frage

"Warum läuft mein Parallelprogramm nicht?"

und nur ganz wenige in der Beantwortung der Frage

"Warum läuft mein Parallelprogramm nicht schneller?".

Dies läßt Fragen der Präferenz zwischen möglichst hohen Geschwindigkeitsgewinnen (die - wie beispielhaft gezeigt wurde - ohnehin höchst fragwürdig sind) und der Frage der Programmierbarkeit paralleler Lösungen in einem ganz anderen Licht erscheinen! Schon die in [Knei82] entwickelten Makro-Datenflußmechanismen wurden in erster Linie aus der Überlegung motiviert, dem Benutzer mehr "Systemunterstützung" bei der Programmentwicklung zu geben.

Der durchgeführte Parallelisierungsansatz wird erst dann an Bedeutung verlieren, wenn für Multiprozessorsysteme die Testsysteme existieren, die heute schon für den Einzelprozessor verfügbar sind. Mit dieser Arbeit ist nicht nur der Wunsch verbunden, Anregungen zu weiteren Forschungen auf dem Gebiet der

Programmierbarkeit paralleler Programme zu geben. Es sollte darüberhinaus ein Einblick in die Probleme gegeben werden, die noch zu lösen sind, bevor der Vorgang der Parallelisierung beliebiger sequentieller Programme für bestimmte Multiprozessor-systeme automatisch erfolgen kann.

Literaturverzeichnis

- [Acke79] Ackerman, W. B.: "Data flow languages", vol. 48, pp. 1087-1095 in AFIPS Proc. of the 1979 Nat. Comp. Conf. (1979).
- [Alt78] Altmann, W.: "Beschreibung von Programm-Moduln zum Entwurf zuverlässiger Software-Systeme", Dissertation, Arbeitsberichte des IMMD, Band 11, Nr. 16, Erlangen (1978).
- [AnSch83] Andrews, G. R., Schneider, F. B.: "Concepts and Notations for Concurrent Programming", Computing Surveys, vol. 15, no. 1, pp. 3-43 (1983).
- [Back78] Backus, J.: "Can Programming be liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Comm. ACM, vol. 21, no. 8, pp. 613-641 (1978).
- [Baer73] Baer, J.: "A survey of some theoretical aspects of multiprocessing", Computing Surveys, vol. 5, no. 1, pp. 31-80 (1973).
- [BCKT79] Banerjee, U., Chen, S. C., Kuck, D. J., Towle, R. A.: "Time and Parallel Processor Bounds for FORTRAN-like Loops", IEEE Transactions on Computers, vol. C-28, no. 9, pp. 660-670 (1979).
- [Bath82] Bathelt, P.: "Vergleich von Synchronisationsmechanismen", Diplomarbeit am IMMD IV, Arbeitsberichte des IMMD, Band 15 Nr. 3, Erlangen (1982).

- [Bern66] Bernstein, A. J.: "Analysis of Programs for Parallel Processing", IEEE Trans. Electr. Comput., vol. EC-15, pp. 757-762 (1966).
- [BlMS83] Blair, G. S., Mariani, J. A., Sheperd, D.: "A Practical Extension to UNIX for Interprocess Communication", Software - Practice and Experience, vol. 13, no. 1, pp. 45-58 (1983).
- [BoHä83] Bode, A., Händler, W.: "Rechnerarchitektur II", Springer-Verlag, Berlin-Heidelberg-New York (1983).
- [CoES71] Coffman, E. G., Elphick, M. J., Shoshani, A.: "System Deadlocks", Computing Surveys, vol. 3, no. 2, pp. 67-78 (1971).
- [Com78] Comte, D., Durrieu, G., Gelly, O., Plas, A., Syre, J. C.: "Parallelism, Control and Synchronization Expression in a Single Assignment Language", SIGPLAN Notices, vol. 13, no. 1, pp. 25-33 (1978).
- [Dem82] Deminet, J.: "Experience with Multiprocessor Algorithms", IEEE Transactions on Computers, vol. C-31, no. 4, pp. 278-288 (1982).
- [Den79] Denert, E.: "Software-Modularisierung", Informatik-Spektrum, vol. 2, pp. 204-218 (1979).
- [Denn80] Dennis, J. B.: "Dataflow Supercomputers", Computer, vol. 13, pp. 48-56 (1980).
- [Dij68] Dijkstra, E. W.: "Cooperating Sequential Processes", pp. 43-112 in Programming Languages, ed. F. Genuys, Academic Press, New York (1968).

- [Dre83] Dreyer, T.: "EGPA-Betriebssystem-Moduln", EGPA-Dokumentation, Band EGPA.SYS, Erlangen (1983).
- [DDJT79] Durham, I., Dugan, R. C., Jones, A. K., Talukdar, S. N.: "Power System Simulation on a Multiprocessor" in IEEE Power Engineering Summer Meeting (1979).
- [Ens77] Enslow, P. H.: "Multiprocessor Architecture - A survey", Computing Surveys, vol. 9, no. 1, pp. 103-129 (1977).
- [Fel79] Feldman, J. A.: "High level programming for distributed computing", Comm. ACM, vol. 22, no. 6, pp. 353-368 (1979).
- [FKLV83] Fritsch, G., Kleinöder, W., Linster, C. U., Volkert, J.: "EMSY85 - The Erlangen Multi-Processor System for a Broad Spectrum of Applications", pp. 325-330 in Proc. of the 1983 International Conference on Parallel Processing, Aug 23-26, 1983, IEEE Comp. Soc. Order No. 479, ed. Siegel H. J., Siegel, L., IEEE Computer Society Press (1983).
- [Fri84] Fritsch, G.: "Memory-coupled processor arrays for a broad spectrum of applications", Lecture Notes in Physics, vol. 196, pp. 158-177 in Proc. WOPLOT 83, Parallel Processing: Logic, Organization, and Technology, Neubiberg, Bavaria, Germany, June 27-29, 1983, ed. Becker, J., Eisele, I., Springer-Verlag, Berlin-Heidelberg-New-York (1984).
- [From82] Fromm, H. J.: "Multiprozessor-Rechenanlagen: Programmstrukturen, Maschinenstrukturen und Zuordnungsprobleme", Dissertation, Arbeitsberichte des IMMD, Band 15 Nummer 5, Erlangen (1982).

- [FHHJ83] Fromm, H. J., Hercksen, U., Herzog, U., John, K.-H., Klar, R., Kleinöder, W.: "Experiences with Performance Measurement and Modeling of a Processor Array", IEEE Transactions on Computers, vol. C-32, no. 1, pp. 15-31 (1983).

- [FuSS82] Furuta, R., Scofield, J., Shaw, A.: "Document Formatting Systems: Survey, Concepts, and Issues", Computing Surveys, vol. 14, no. 3, pp. 417-472 (1982).

- [GJS83] Gehringer, E. F., Jones, A. K., Segall, Z. Z.: "The CM* Testbed", Computer, vol. 15, pp. 40-53 (1982).

- [Ger77] Gerber, A. J.: "Process Synchronization by Counter Variables", Operating Systems Review, vol. 11, no. 4, pp. 6-17 (1977).

- [Gilo82] Giloi, W. K.: "Datenflußprinzipien in Rechnerarchitekturen", NTG-Fachberichte, vol. 80, pp. 19-41 in Struktur und Betrieb von Rechensystemen, VDE-Verlag, Berlin-Offenbach (1982).

- [GiGu82] Giloi, W. K., Güth, R.: "Das Prinzip der Datentypen-Architekturen und seine Realisierung im STARLET-Rechner", Informatik-Spektrum, vol. 1, pp. 21-37 (1982).

- [GoRa72] Gonzalez, M. J., Ramamoorthy, C. V.: "Parallel Task Execution in a Decentralized System", IEEE Transactions on Computers, vol. C-21, no. 12, pp. 1310-1322 (1972).

- [Gon77] Gonzalez, M. J.: "Deterministic Processor Scheduling", Computing Surveys, vol. 9, no. 3, pp. 173-204 (1977).

- [Goo75] Goodenough, J. B.: "Exception Handling: Issues and a Proposed Notation", Comm. ACM, vol. 18, no. 12, pp. 683-696 (1975).
- [GoKa78] Goos, G., Karstens, U.: "Programming languages and the design of modular programs", pp. 153-186 in Constructing quality software, ed. Hibbard, P. G., Schuman, S. A., Amsterdam, North Holland (1978).
- [GrTa82] Grosser, I., Talukdar, S. N.: "Models for MIMD Machines", IEEE Transactions on Power Apparatus and Systems, vol. PAS - 101, pp. 94-99 (1982).
- [GuHo78] Guttag, J. V., Hornig, J. J.: "The algebraic specification of abstract data types", Acta Informatica, vol. 10, pp. 27-52 (1978).
- [Hän73] Händler, W.: "The concept of Macro-Pipelining with high availability", Elektronische Rechenanlagen, vol. 15, no. 6, pp. 269-274 (1973).
- [HHS76] Händler, W., Hofmann, F., Schneider, H.J.: "A General Purpose Array with a Broad Spectrum of Applications", vol. 4, pp. 311-335 in Informatik-Fachberichte, ed. Händler, W., Springer-Verlag, Berlin-Heidelberg-New York (1976).
- [HäRo80] Händler, W., Rohrer, A.: "Gedanken zu einem Rechner-Baukasten-System", Elektronische Rechenanlagen, vol. 22, no. 1, pp. 3-13 (1980).
- [Hän82] Händler, W.: "Innovative Computer Architecture - How to enlarge parallelism and not complexity", pp. 1-41 in Parallel Processing Systems, 1980 Proc. Symp. Loughborough Univ. Techn., ed. Evans, D. J., Univ. Press, Cambridge, England (1982).

- [HHHS83] Händler, W., Herzog, U., Hofmann, F., Schneider, H. J. und Mitarbeiter: "Projektabschlußbericht 1978 - 82 EGPA Erlangen General Purpose Array,", Arbeitsberichte des IMMD, Band 16 Nr. 6, pp. 1-49, Erlangen (1983).
- [HBFHV84] Händler, W., Bode, A., Fritsch, G., Henning, W., Volkert, J.: "A tightly coupled and hierarchical multiprocessor architecture" in Proc. Conf. on Vector and Parallel Processors in Computational Science II, Aug 28-31, 1984, Oxford (1984).
- [Hai84] Haider, L.: "Entwurf und Implementierung einer Interprozessorkommunikation für das DIRMU-Multiprozessor-system in der Programmiersprache MODULA-2", Diplomarbeit am IMMD III, Erlangen (1984).
- [Han72] Brinch Hansen, P.: "A Comparison of Two Synchronizing Concepts", Acta Informatica, vol. 1, pp. 190-199 (1972).
- [Han73] Brinch Hansen, P.: "Concurrent Programming Concepts", Computing Surveys, vol. 5, no. 4, pp. 223-245 (1973).
- [Han78] Brinch Hansen, P.: "Distributed Processes: A Concurrent Programming Concept", Comm. ACM, vol. 21, no. 9, pp. 934-941 (1978).
- [HaWu78] Harbison, S. P., Wulf, W. A., CMU: "Reflections in a pool of processors: An Experience Report on C.mmp/Hydra", pp. 939-951 in Proc. AFIPS Nat. Comp. Conf. (1978).

- [Hoa72] Hoare, C. A. R.: "Towards a Theory of Parallel Programming", pp. 61-71 in Operating System Techniques, Academic Press, New York (1972).
- [Hoa74] Hoare, C. A. R.: "Monitors: An Operating System Structuring Concept", Comm. ACM, vol. 17, no. 10, pp. 549-557 (1974).
- [Hoa78] Hoare, C. A. R.: "Communicating Sequential Processes", Comm. ACM, vol. 21, no. 8, pp. 666-677 (1978).
- [Hof84] Hofmann, F.: "Warum Interrupts?", Persönliche Mitteilung, Erlangen (1984).
- [Hol72] Holt, R. C.: "Some deadlock properties of Computer Systems", Computing Surveys, vol. 4, no. 3, pp. 179-196 (1972).
- [Hoppe84] Hoppe, B.: "Deterministisches Zuordnen vielfach durchlaufener Aufgabensysteme in Multiprozessorsystemen", Dissertation, Arbeitsberichte des IMMD, Band 17 Nummer 6, Erlangen (1984).
- [Horn84] Horn, H.-J.: "Analyse des Informationsflusses in der EGPA-Pyramide mit Hilfe des Meßmonitors ZM III", Diplomarbeit am IMMD VII, Erlangen (1984).
- [HoVa84] Hosemann, G., Viegas de Vasconcelos, A.: "Transient Studies on a Multiprocessor", IEEE Transactions on Power Apparatus and Systems, vol. PAS-103, no. 11, pp. 3260-3266 (1984).

- [HosW83] Hoßfeld, F., Weidner, P.: "Parallele Algorithmen", Informatik-Spektrum, vol. 6, pp. 142-154 (1983).
- [Iss83] v. Issendorff, H.: "Netzwerkprogrammierung - Eine universelle Programmiermethodik", Forschungsinstitut für Funk und Mathematik, Forschungsgesellschaft für Angewandte Naturwissenschaften, Bericht Nr. 328, Wachtberg-Werthoven (1983).
- [JaSt77] Jammel, A. J., Stiegler, H. G.: "Managers versus Monitors", pp. 827-830 in Information Processing 77, ed. Gilchrist, B., Amsterdam, North Holland (1977).
- [Jon80] Jones, A. K., et al.: "The CM* Multiprocessor Project: A research review", Technical Report CMU-80-131, ed. Jones, A. K., Gehring E. F., Computer Science Department, Carnegie-Mellon-University (1980).
- [JoSch80] Jones, A. K., Schwarz, P.: "Experience using multiprocessor systems: a status report", Computing Surveys, vol. 11, no. 2, pp. 121-165 (1980).
- [KaMQ77] Kahn, G., MacQueen, D. B.: "Coroutines and Networks of Parallel Processes", pp. 993-998 in Information Processing 77, ed. Gilchrist, B., Amsterdam, North Holland (1977).
- [KaKo83] Kammerer, A., Konopik, H.: "Auswertung von Spektren und Betrachtungen über eine mögliche Nutzung einer Multiprozessor-Rechenanlage", Diplomarbeiten am IMMD III, Erlangen (1983).

- [Ker82] Keramidis, S.: "Eine Methode zur Spezifikation und korrekten Implementierung asynchroner Systeme", Arbeitsberichte des IMMD, Band 15 Nr. 4, Erlangen (1982).
- [KeRi78] Kernighan, B. W., Ritchie, D. M.: "The C Programming Language", Prentice-Hall (1978).
- [Klei82] Kleinöder, W.: "Stochastische Bewertung von Aufgabenstrukturen für hierarchische Mehrrechnersysteme", Dissertation, Arbeitsberichte des IMMD, Band 15 Nummer 10, Erlangen (1982).
- [Knei80] Kneißl, F.: "Eine Stack/String/Listenmaschine für die Textverarbeitung", pp. 32-49 in GI-Tagung Textverarbeitung und Informatik, 28.5 - 30.5.1980, Bayreuth (1980).
- [Knei82] Kneißl, F.: "Realisierung von Makro-Datenflußmechanismen auf hierarchischen Mehrrechnersystemen", Dissertation, Arbeitsberichte des IMMD, Band 15 Nummer 11, Erlangen (1982).
- [KnRa82] Kneißl, F., Rathke, M.: "TEXT - Ein System zur Herstellung und Pflege von Dokumenten", EGPA-Dokumentation, Band EGPA.TEX, Erlangen (1982).
- [KKK76] Kober, R., Kopp, H., Kuznia, Ch.: "SMS-101 - A structured Multimicroprocessor-System with deadlock free operation scheme", Euromicro Newsletter, no. 2, pp. 56-64 (1976).
- [KNU75] Krayl, H., Neuhold, E. J., Unger, C.: "Grundlagen der Betriebssysteme", Sammlung Götschen, Walter de Gruyter, Berlin-New York (1975).

- [KMC72] Kuck, D. J., Muraoka, Y., Chen, S. C.: "On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and their Resulting Speedup", IEEE Transactions on Computers, vol. C-21, no. 12, pp. 1293-1310 (1972).

- [Kuck77] Kuck, D. J.: "A Survey of Parallel Machine Organization and Programming", Computing Surveys, vol. 9, no. 1, pp. 29-59 (1977).

- [Kung76] Kung, H. T.: "Synchronized and asynchronous parallel algorithms for multiprocessors", pp. 153-200 in Algorithms and Complexity - New Directions and Results, ed. Traub, F. J., Academic Press, New York (1976).

- [Kung80] Kung, H. T.: "The Structure of Parallel Algorithms", vol. 19, pp. 65-112 in Advances in Computers, ed. Yovits, M. C., Academic Press (1980).

- [Lam74] Lamport, L.: "The Parallel Execution of DO-Loops", Comm. ACM, vol. 17, no. 2, pp. 83-93 (1974).

- [LaNe79] Lauer, H. C., Needham, R. M.: "On the Duality of Operating System Structures", Operating Systems Review, vol. 13, no. 2, pp. 3-19 (1979).

- [Lee83] Lee, P. A.: "Exception Handling in C Programs", Software - Practice and Experience, vol. 13, no. 5, pp. 389-405 (1983).

- [Lin81] Linster, C. U.: "SYMPOS/UNIX - Ein Betriebssystem für homogene Polyprozessorsysteme", Dissertation, Arbeitsberichte des IMMD, Band 14 Nr. 3, Erlangen (1981).

- [LiAg81] Lint, B., Agerwala, T.: "Communication Issues in the Design and Analysis of Parallel Algorithms", IEEE Transactions on Software Engineering, vol. SE-7, no. 2, pp. 174-188 (1981).
- [LiZi74] Liskov, B. H., Zilles, S.: "Programming with Abstract Data Types", SIGPLAN Notices, vol. 9, no. 4, pp. 50-59 (1974).
- [LiSn77] Liskov, B. H., Snyder, A., Atkinson, R., Schaffert, C.: "Abstraction mechanisms in CLU", Comm. ACM, vol. 20, no. 8, pp. 564-576 (1977).
- [Lis79] Liskov, B. H.: "Primitives for distributed Computing", pp. 33-42 in Proc 7th Symp. Operating Systems Principles, Pacific Grove, Calif., Dec 10-12, 1979, ACM, New York (1979).
- [LiSn79] Liskov, B. H., Snyder, A.: "Exception Handling in CLU", IEEE Transactions on Software Engineering, vol. SE-5, no. 6, pp. 546-558 (1979).
- [Lis82] Liskov, B. H.: "On Linguistic Support for Distributed Programs", IEEE Transactions on Software Engineering, vol. SE-8, no. 3, pp. 203-210 (1982).
- [MacL77] MacLaren, M. D.: "Exception Handling in PL/I", SIGPLAN Notices, vol. 12, no. 3, pp. 101-104 (1977).
- [MeRa77] Melliar-Smith, P. M., Randell, B.: "Software reliability: The role of programmed exception handling", SIGPLAN Notices, vol. 12, no. 3, pp. 95-100 in Proc ACM Conf. on Language Design for Reliable Software (1977).

- [Mes82] Messerer, M.: "Ein neuer Ansatz zur Parallelisierung von Compilern", Dissertation, Arbeitsberichte des IMMD, Band 15 Nr. 8, Erlangen (1982).
- [Müh84] Mühlen, V.: "Ereignisorientiertes Meßsystem", Diplomarbeit am IMMD VII, Erlangen (1984).
- [My75] Myers, G. J.: "Reliable Software through composite design", Petrocelli/Charter, New York (1975).
- [OwiG76] Owicki, S., Gries, D.: "Verifying Properties of Parallel Programs: An Axiomatic Approach", Comm. ACM, vol. 19, no. 5, pp. 279-285 (1976).
- [Par72] Parnas, D. L.: "On the criteria to be used in decomposing systems into modules", Comm. ACM, vol. 15, no. 12, pp. 1053-1058 (1972).
- [Rama69] Ramamoorthy, C. V., Gonzalez, M. J.: "A survey of techniques for Recognizing Parallel Processable Streams in Computer Programs", vol. 35, pp. 1-15 in FJCC Proc. (1969).
- [RaChG72] Ramamoorthy, C. V., Chandy, K. M., Gonzalez, M. J.: "Optimal Scheduling Strategies in a Multiprocessor System", IEEE Transactions on Computers, vol. C-21, no. 2, pp. 137-146 (1972).
- [RaKi74] Ramamoorthy, C. V., Kim, K. H.: "Pipelining - The Generalized Concept and Sequencing Strategies", pp. 289-297 in Proc. AFIPS National Computer Conference (1974).

- [RaLi74] Ramamoorthy, C. V., Li, H. F.: "Efficiency in Generalized Pipeline Networks", pp. 625-635 in Proc. AFIPS National Computer Conference (1974).
- [RaLi77] Ramamoorthy, C. V., Li, H. F.: "Pipeline Architecture", Computing Surveys, vol. 9, no. 1, pp. 61-102 (1977).
- [Rat83] Rathke, M.: "Implementierung und Validierung paralleler Algorithmen auf Monorechnern", Angewandte Informatik, vol. 8, pp. 337-344 (1983).
- [ReKa79] Reed, D. P., Kanodia, R. K.: "Synchronization with Eventcounts and Sequencers", Comm. ACM, vol. 22, no. 2, pp. 115-123 (1979).
- [RiTh74] Ritchie, D. M., Thompson, K. L.: "The UNIX Time-sharing System", Comm. ACM, vol. 17, no. 7, pp. 365-375 (1974).
- [Robi79] Robinson, J. T.: "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms", IEEE Transactions on Software Engineering, vol. SE-5, no. 1, pp. 24-31 (1979).
- [Rou77] Roucairol, G. P.: "On Parallelization of "Single Assignment" Programs", pp. 203-206 in Parallel Computers - Parallel Mathematics, ed. Feilmeier, M., North Holland Publishing Company (1977).
- [Swob75] Swoboda, J.: "Was ist Pipelining?", Elektronische Rechenanlagen, vol. 17, no. 2, pp. 80-83 (1975).
- [Trel79] Treleaven, P. C.: "Exploiting Program Concurrency in Computing Systems", Computer, vol. 12, no. 1, pp. 42-50 (1979).

- [TrBH82] Treleaven, P. C., Brownbridge, D. R., Hopkins, R. P.: "Data-driven and demand-driven computer architecture", Computing Surveys, vol. 14, no. 1, pp. 93-143 (1982).
- [Vas83] Viegas de Vasconcelos, A.: "Sequentielles Programm zur Analyse elektrischer Netze", Persönliche Mitteilung, Erlangen (1983).
- [Vas84] Viegas de Vasconcelos, A.: "Multiprozessoren und ihre Anwendung zur Analyse elektrischer Netze", Elektronische Rechenanlagen, vol. 26, no. 3, pp. 145-151 (1984).
- [Wal82] Wallach, Y.: "Alternating Sequential/Parallel Processing", Lecture Notes in Computer Science, vol. 127, Springer-Verlag, Berlin-Heidelberg-New York (1982).
- [Web83] Weber, K.: "Modellierung von Fehlverhalten mit Berücksichtigung paralleler Abläufe", Dissertation, Arbeitsberichte des IMMD, Band 16 Nr. 9, Erlangen (1983).
- [Wen78] Wendler, K.-D.: "Betriebssystemaspekte in hierarchisch modularen Polyprozessorsystemen - Modellierungsansätze und Koordinierungsmechanismen", Dissertation, Arbeitsberichte des IMMD, Band 11 Nummer 15, Erlangen (1978).
- [Win79] Winkler, J. F. H.: "Das Prozeßkonzept in Betriebssystemen und Programmiersprachen I", Informatik-Spektrum, vol. 2, pp. 219-229 (1979).
- [Win80] Winkler, J. F. H.: "Das Prozeßkonzept in Betriebssystemen und Programmiersprachen II", Informatik-Spektrum, vol. 3, pp. 31-40 (1980).

- [WLS76] Wulf, W. A., London, R. L., Shaw, M.: "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology", Techn. Report, Carnegie-Mellon-University, Dep. of Computer Science (1976).
- [Zisch83] Zischler, H.: "Einführung in die Programmierung der EGPA-Anlage", Arbeitsberichte des IMMD, Band 16 Nr. 6, pp. 76-130, Erlangen (1983).

Stichwortverzeichnis

- A**abhängigkeit (von Aufgaben) 22
adaptive Implementierung 100, 138
aktiv warten 164
Alarmmarke 193
Analyseprozeß 185
Anlaufphase 93
anteilsweise Berechnung 133
APP (alternating parallel/parallel) 81
Arbeitsprozessoren 79
ASP (alternating sequential/parallel) 80
asynchrone Weitergabe von Daten 74
asynchrones Starten von Prozessoren 81
Ausgaben so früh wie möglich 46, 167
ausgabeorientierte Strategie 75
Ausgabeprozessor 104
Ausgabespeicher (-puffer) 104
Auslaufphase 94
Auslaufzustand 92
- B**aumhöhenreduktion 29
Blockiertbedingung (beliebig) 184
Blockiertbedingung (normiert) 180
botschaftsorientiert 162
- C**omposite Design 38
- D**atenfluß 23
Datenflußmechanismen 27
Datenströme 39
Deadlock (allgemein) 170
Deadlock (in Betriebssystemen) 172
dezentrale Organisation 105
DFC-Dateien (Data Flow Control) 155
- E**/A-orientiert 40
Effizienz 107
EGPA, EGPA-Pilotpyramide 5, 17
Eingaben so spät wie möglich 46, 167
eingabeorientierte Strategie 75
Eingabeprozessor 104
Eingabespeicher (-puffer) 104
Einschwingzustand 92
Entflechtung 60
Ereigniszähler 164
- F**IFO-Puffer 150
- G**eschwindigkeitsgewinn 4
Granularität 22
- H**ierarchie 16
Hilfsmodul 68
horizontale Verarbeitungsrichtung 57
Hyperabsatz 119
- ideales Rechensystem 28
informationskonsistent 55
Interrupt 164
- K**ernphase 93
Kernzustand 92
Kommunikationsmodul 68
Kommunikationsprozeduren 40
Kommunikationsspeicher 78, 205
Komplexitätsmaß $O(n)$ 29
komponentenweise Berechnung 133
Konnektivitätsmatrix 25
Kontrollfluß 23
Kontrollprozessor 79
kritischer Pfad (eines Aufgabengraphen) 84, 99
- L**esekonflikt 177
Listenspeicher 152
- M**akro-Datenfluß 147
minimale Arbeitsmenge 55
Modularisierung 37
Multiprozessorsystem (hierarchisch) 15
- N**achbarhilfe 137
nebenläufig 35
Netzwerkanalyse 125

- Ordnung** $O(n)$ 29
- ordnungserhaltende Programmsysteme 6, 14
- ordnungsfreie Programmsysteme 14
- Parallel** ausführbar 22
- parallele Programmierungsumgebung 3, 19
- parallele Prozesse 34
- Parallelprogramm 36
- partieller Deadlock 172
- pipelineartig 35
- Pipelining 71
- Pipes 146
- Präzedenzgraph 25
- Programmierbarkeit (Kriterien) 1
 - prozedurorientiert 162
- Prozedurquittungen 43
- Prozessor 15
- Prozessorelement 15
- Round-Robin-Heuristik** 92
- Schreibkonflikt** 177
- single assignment 24
- speicherorientierte Heuristik 98
- Strategiemodul 67
- synchrone Weitergabe von Daten 74
- synchrones Starten von Prozessoren 81
- Synchronisationsmodul 161
- synchronisierter Informationsaustausch 214
- Systemverklemmung (allgemein) 170
- Systemverklemmung (in Betriebssystemen) 172
- Teilaufgabe** (E/A-orientiert mit/ohne Kontrolle) 40, 43
- Textformatierung 109
- Timeout-Mechanismus 191
- totaler Deadlock 173
- Transformationsmodul 67
- Verfügbarkeit** von Operanden 24
- verkettete Listen 152
- vertikale Verarbeitungsrichtung 57
- verzögert aktiv warten 168
- Wellenfront-Methode** 32
- zustandsorientierte Heuristik** 96

Lebenslauf Martin Rathke

- 3.4.1953 geboren in Minden/Westf., Vater Dr. Friedrich Wilhelm Rathke, Arzt, Mutter Ursula Rathke
- 1960-1972 Volksschulen Heidelberg und Ludwigsburg, Schiller-Gymnasium Ludwigsburg, neusprachliches Gymnasium mit Fremdsprachen Englisch, Latein, Französisch
- 1972 Abitur
- 1972-1974 Ausbildung zum "Staatlich geprüften Programmierer" an der Fachhochschule Konstanz
- 1974-1976 Studium der Informatik ab 3. Semester in Stuttgart bis zum Vordiplom
- 1976-1978 Studium der Informatik in Erlangen, Nebenfach Mathematik, Schwerpunktfach Betriebssysteme
- 1978 Diplomprüfung mit Gesamturteil "sehr gut", Diplomarbeit bei Dr. Keramidis "Methoden zur Konstruktion zuverlässiger Systemsoftware"
- seit 1979 Wiss. Mitarbeiter am IMMD III der Univ. Erlangen-Nürnberg im Rahmen des EGPA-Projekts.
Software für Graphik und Textverarbeitung, Programmierungsumgebungen für Multiprozessorsysteme, Multiprozessor-Anwendungen unter den Systemen EGPA und UNIX, organisatorische Aufgaben im EGPA-Projekt, Benutzerberatung

In letzter Zeit sind folgende Arbeitsberichte des Instituts für
Mathematische Maschinen und Datenverarbeitung erschienen:

Band 13

- =====
- Nr. 1 Schrammel, D., Bayer, W., Gliemann, H.-P., Heigl, M.,
Mertens, P.:
Ein Leistungs- und Wirtschaftlichkeitsvergleich zwischen
Klein- und Großcomputern
(Januar 1980)
- Nr. 2 Göttler, H.:
Semantische Modelle
(März 1980)
- Nr. 3 Fromm, H.J.:
Zur Modellierung der Speicherinterferenz bei hierarchisch
organisierten Multiprozessorsystemen
(April 1980)
- Nr. 4 Jäpel, D.:
Klassifikatorbezogene Merkmalsauswahl
(Juli 1980)
- Nr. 5 Bodendorf, F., Mertens, P., Haas, R., Weber, G., Wolf, G.:
SAMBA - ein Methodenbankrahmen um das Statistik-Paket SPSS
(Juli 1980)
- Nr. 6 Bartsch, B.:
Inferenz und Analyse spezieller Graphgrammatiken für die
syntaktische Mustererkennung
(September 1980)
- Nr. 7 Wurm, F.X.:
Auftragssystem für eine Mehrprozessoranlage
(Oktober 1980)
- Nr. 8 Schedelbeck, A.:
Vergleich der beiden höheren Realzeitprogrammiersprachen
ADA und PEARL
(Oktober 1980)
- Nr. 9 Jahresbericht 1980 der Informatik
(März 1981)

- Nr. 1 Pattern Recognition
 (Research at Lehrstuhl für Informatik 5)
 (Februar 1981)
- Nr. 2 Bunke, H.:
 Analyse elektrischer Schaltpläne
 (März 1981)
- Nr. 3 Linster, C.-U.:
 SYMPOS/UNIX - Ein Betriebssystem für homogene
 Polyprozessorsysteme
 (Juni 1981)
- Nr. 4 Akyildiz, J.F., Bolch, G.:
 Analytic Solution Techniques for Queueing Network
 Models of Computer Systems
 (Juli 1981)
- Nr. 5 Lindstedt, W.G.:
 Ein Verfahren zur Erstellung portabler algorithmischer
 Laufzeitprogramme
 (Juli 1981)
- Nr. 6 Vollmar, R.:
 Zum Begriff des Parallelismus bei Polyautomaten
 (Juli 1981)
- Nr. 7 Luft, A.L.:
 Rationaler Sprachgebrauch und orthosprachliche
 Standardisierung als Grundlagen zuverlässiger Software-
 Entwicklung und -Dokumentation
 (Januar 1982)
- Nr. 8 Händler, W.:
 PROCEEDINGS OF A WORKSHOP ON TAXONOMY IN COMPUTER
 ARCHITECTURE
 (Februar 1982)
- Nr. 9 Beth, Th., Hess, P., Wirl, K.:
 Materialien zur Kryptographie
 (März 1982)
- Nr. 10 Jahresbericht 1981 der Informatik
 (März 1982)

- Nr. 1 Wittmann, A.:
Ein Mechanismus für die Synchronisation paralleler
Prozesse
(Februar 1982)
- Nr. 2 Maehle, E.:
Fehlertolerantes Verhalten in Multiprozessoren -
Untersuchungen zur Diagnose und Rekonfiguration
(März 1982)
- Nr. 3 Bathelt, P.:
Vergleich von Synchronisationsmechanismen
(Mai 1982)
- Nr. 4 Keramidis, S.:
Eine Methode zur Spezifikation und korrekten
Implementierung von asynchronen Systemen
(Juni 1982)
- Nr. 5 Fromm, H.J.:
Multiprozessor-Rechenanlagen:
Programmstrukturen, Maschinenstrukturen und Zuordnungs-
probleme
(Mai 1982)
- Nr. 6 Bley, H.:
Vorverarbeitung und Segmentierung von Stromlaufplänen
unter Verwendung von Bildgraphen
(Juli 1982)
- Nr. 7 Hohl, W.:
Informatik-Sammlung
Katalog mit historischen Erläuterungen
(August 1982)
- Nr. 8 Messerer, M.:
Ein neuer Ansatz zur Parallelisierung von
Compilern
(August 1982)
- Nr. 9 Grosch, J.:
Eine Programmiersprache mit mengentheoretischen
Konstrukten und deren effiziente Implementierung
(August 1982)
- Nr. 10 Kleinöder, W.:
Stochastische Bewertung von Aufgabenstrukturen
für hierarchische Mehrrechnersysteme
(August 1982)

Band 15 Fortsetzung

=====

- Nr. 11 Kneißl, F.:
Realisierung von Makro-Datenflußmechanismen auf
hierarchischen Mehrrechnersystemen
(August 1982)
- Nr. 12 Brendel, W., Fan, Z., Klar, R., Schmielau, W.:
ERES 82
Handbuch und Fallstudie
Handbook and Case Study
(Oktober 1982)
- Nr. 13 Pelz, K.:
Portable Codegenerierung in Übersetzern
für Prozeß-Programmiersprachen
(Oktober 1982)
- Nr. 14 Sommer, U., Steinchen, J.:
Ein Kleinrechner-Grafiksystem zur Dokumentation
von EDV-Abläufen
(Oktober 1982)
- Nr. 15 Hein, H.-W.:
Das Erlanger Spracherkennungssystem
(November 1982)
- Nr. 16 Bolch, G., Bruchner, W.:
Analytische Modelle für symmetrische Mehrprozessor-
anlagen mit dynamischen Prioritäten
(Dezember 1982)
- Nr. 17 Jahresbericht 1982 der Informatik
(März 1983)

- Nr. 1 Gall, R.:
Formale Beschreibung des inkrementellen Programmierens-
im-Großen mit Graph-Grammatiken
(März 1983)
- Nr. 2 König, R.:
Beiträge zur algebraischen Theorie der formalen Sprachen
(März 1983)
- Nr. 3 Welbourn, D.B.:
The Design of Mechanical Components and the Development
of DUCT:
17 Years of CAD/CAM in Cambridge University
(April 1983)
- Nr. 4 Woitok, R.:
Die System-Implementierungs-Sprache ALICE
(Mai 1983)
- Nr. 5 Schneider, S., Schwab, P., Renninger, W.:
Wesen, Vergleich und Stand von Software zur Produktion
von Systemen der computergestützten Unternehmensplanung
(Juli 1983)
- Nr. 6 Händler, W., Herzog, U., Hofmann, F., Schneider, H.J.
und Mitarbeiter:
Projektabschlußbericht 1978 - 1982 Erlangen General Purpose
Array
(Juli 1983)
- Nr. 7 Mackert, L.:
Modellierung, Spezifikation und korrekte Realisierung
von asynchronen Systemen
(Juli 1983)
- Nr. 8 Reitenspieß, M.:
Sprachkonstrukte zur Spezifikation und korrekten Implemen-
tierung von Schutzproblemen
(Juli 1983)
- Nr. 9 Weber, K.:
Modellierung von Fehlverhalten mit Berücksichtigung
paralleler Abläufe
(August 1983)
- Nr. 10 Dietsch, H.:
Ein verteiltes Kommunikationssystem für einen Verbund
aus lokalen Mikrorechner-Netzwerken
(Dezember 1983)

Band 16 Fortsetzung

=====

- Nr. 11 Maehle, E., Schmitter, E.:
 Workshop Fehlertolerante Mehrprozessor- und Mehr-
 rechnersysteme
 (Dezember 1983)
- Nr. 12 Volkert, J.:
 Multiplikative Abstandsfunktionen und deren Anwendung
 in der Bildverarbeitung
 (Dezember 1983)
- Nr. 13 Steinbauer, D.:
 Transaktionen als Grundlage zur Strukturierung und
 Integritätssicherung in Datenbank-Anwendungssystemen
 (Dezember 1983)
- Nr. 14 Wilke, P.:
 Zusammenhänge und Unterschiede zwischen Graph-Grammatiken
 und Petri-Netzen sowie verwandter Systeme
 (Dezember 1983)
- Nr. 15 Jahresbericht der Informatik 1983
 (März 1984)

Band 17

=====

- Nr. 1 Eberlein, W.:
Architektur technischer Datenbanken für
integrierte Ingenieursysteme
(Februar 1984)
- Nr. 2 Mertens, P., Heigl, M.:
Neuere Entwicklungen der computergestützten
Produktionsplanung, Eignung - Verbindungen -
Entwicklungspfade
(April 1984)
- Nr. 3 Stoyan, H.:
Maschinen- unabhängige Code-Erzeugung als semantik-
erhaltende beweisbare Programmtransformation
(Juli 1984)
- Nr. 4 Brendel, W.:
Funktionaler Entwurf und automatische Synthese
hochintegrierter Schaltkreise
(August 1984)
- Nr. 5 Brietzmann, A.:
Semantische und pragmatische Analyse im Erlanger
Spracherkennungsprojekt
(Oktober 1984)
- Nr. 6 Hoppe, B.:
Deterministisches Zuordnen vielfach durchlaufener
Aufgabensysteme in Multiprozessorsystemen
(Oktober 1984)
- Nr. 7 Fumy, W., Posch, S., Rieß, H.P.:
Materialien zur Codierungstheorie III
(Dezember 1984)
- Nr. 8 Akyildiz, J. F.:
Leistungsanalyse von Multiprozessorsystemen mit
Prozeßkommunikation
(Dezember 1984)
- Nr. 9 Jahresbericht der Informatik 1984
(März 1985)

Nr. 1 Siegmann, H.:

Wesen, Vergleich und Stand von zehn ausgewählten
kleinrechnerorientierten Planungssprachen
(Februar 1985)

